On the Generation of Quadrilateral Element Meshes for General CAD Surfaces

by

Antonio da Silva Castro Sobrinho

Engenheiro Mecânico, Pontifícia Universidade Católica do Rio de Janeiro, Brasil (1993)

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1996

Author ....
Department of Mechanical Engineering
September, 1996

Certified by ....
Klaus-Jürgen Bathe
Professor of Mechanical Engineering
Thesis Supervisor

Accepted by ....
Ain Ants Sonin
Chairman, Departmental Graduate Committee

On the Generation of Quadrilateral Element Meshes for General CAD Surfaces

by

Antonio da Silva Castro Sobrinho

Engenheiro Mecânico, Pontifícia Universidade Católica do Rio de Janeiro, Brasil (1993)

Submitted to the Department of Mechanical Engineering
on September, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

Abstract

This thesis deals with the automatic generation of meshes of quadrilateral elements over surfaces of a general solid using an advancing front approach. The algorithm is based on *paving*, a method proposed by Blacker *et al.* to mesh planar surfaces. We adopt a CAD integration approach and use AutoCAD-12 to generate the solid and the AutoCAD Development System (ADS) as the interpreter and environment to code the algorithms. The ADS environment, namely the Application Programming Interface library (API), provides the geometric functions to work with the surface during the quadrilateral mesh generation. Finally, we investigate a three-dimensional version named *plastering* for automatic hexahedral mesh generation. The investigation portrays some limitations of the method which leads us to propose a scheme to discretize a general geometry with hexahedral elements employing grid superposition.

Thesis Supervisor: Klaus-Jürgen Bathe
Title: Professor of Mechanical Engineering

to God

Acknowledgments

I thank my wife, Jacqueline, and my daughter, Silvana, for the faithful support I received since the very beginning of this journey. Without their constant dedication I would never have been able to accomplish this research.

A very special extension of gratitude goes to Professor Klaus-Jürgen Bathe for his work as my advisor.

Last, but not least, I thank my parents for the sacrifices they have borne to ensure the fulfillment of my dreams.

Contents

List of Figures

Chapter 1

Introduction

The complexity of the problems solved with the finite element method has demanded automatic mesh generation in the pre-processing stage of the analysis. The mesh has to describe the geometry of the domain and be fine enough to capture the numerical solution in regions where it is not smooth. This latter requirement demands the employment of mesh grading, since coarse discretizations are also necessary in regions of smooth distributions of the solution. Moreover, the elements generated must not be too distorted in order to preserve the accuracy of the solution, especially in sectors of transitions between two different densities [1],[2],[3],[4].

The mesh generation methods can be classified into two main groups: structured and unstructured. Structured methods are algebraic generators based on mapping transformations between a natural and a physical domain. In these schemes, the nodal points are created following algebraic equations associated with the coordinate system adopted. Unstructured methods, such as advancing front and grid superposition, use the geometry as reference to generate the mesh. They are quite independent from the coordinate system in which the geometry is represented, and tend to construct meshes with nodal points randomly generated. Despite the complexity of the algorithms demanded to implement these methods, the level of automation achieved is quite high.

Triangular and tetrahedral mesh generation methods are suitable to automatically discretize complex geometries when implemented with unstructured schemes. These elements, however, perform poorly in their linear displacement version and only their quadratic form can achieve the performance of their counterpart isoparametric quadrilateral and hexahedral elements in problems of elasticity [5],[6]. Quadrilateral and hexahedral elements have been used in transfinite interpolation or isoparametric approaches. These structured methods create smooth meshes and provide good control of the aspect ratio of the elements throughout the domain in both two-dimensional [7],[8],[9] and three-dimensional problems [10], [11], [12], [13],[14]; however, they depend on geometric decomposition techniques to mesh complex configurations with fully automated schemes [15], [16], [17], [18], [19], [20].

In the light of this background, some unstructured mesh generation methods have been published in order to take advantage of the automation of their algorithms and the good performance of the quadrilateral elements. Actually, these methods create quadrilateral elements by combining triangles generated with well-known unstructured schemes, such as advancing front and grid superposition [21], [22], [23], [24], [25], [26]. These approaches, however, tend to provide poor mesh orthogonality along the boundaries where good elements are strongly desired. The combination of tetrahedrons has also been mentioned as a possible extension of these methods to tackle three-dimensional problems [17].

Recently, a new advancing front method using quadrilateral elements was developed by Blacker and Stephenson *et al.* [27], [28], [29], [30] to mesh planar surfaces. Named *paving*, the method starts by creating elements along the boundaries and projecting the fronts inward. This kind of boundary offsetting generates quadrilateral elements with good aspect ratio along the geometric contour. Its three-dimensional version, *plastering*, is based on a similar scheme and has been used to mesh simple geometries [31],[32].

This thesis extends *paving* to automatically generate meshes over the surfaces of general solids created in AutoCAD. We modify the basic structure of the algorithm proposed by Blacker and Stephenson in [30] and introduce the circular advance of the front which increases the speed of generation of elements in the front and, consequently, reduces the time to discretize the entire domain. This modification plays an important role in the overall performance of the method. It counterbalances the poor performance of the routine necessary to project any nodal point created or modified onto the curved surfaces of the solid, and the one used to evaluate the normal vector at the same points. Note these routines are not used if only planar surfaces are considered as in the method proposed by Blacker and Stephenson.

In the second part of this text, we implement an algorithm to mesh simple geometries with *plastering* in order to evaluate the possibilities and limitations of the method. Finally, taking into consideration the results of this evaluation, we investigate different approaches to mesh general solid geometries with hexahedral and non-hexahedral elements, and point out the one that maximizes the number of good hexahedrons generated in the discretization and that best fulfills the characteristics desired for finite element analysis.

Chapter 2

Automatic Quadrilateral Mesh Generation: A Review

As part of the pre-processing stage, indispensable for any finite element analysis, the mesh generation algorithms have demanded an increasing level of automation in order to avoid the errors made during the discretization of the physical domain, and reduce the time consumed in this process. One of the first automatic methods used for quadrilateral mesh generation generated structured meshes and required domains with simple geometric shapes that could be mapped to Cartesian natural coordinate systems. The predominance of this method can be clearly noted in the work of Buell and Bush published in 1972, in which this technique is described under the title of *I-J Transformation* [33]. This technique is still widely used in commercial finite element packages that take advantage of the relatively easy computational implementation to shorten the pre-processing time. Commonly referred to as *algebraic mesh generation* or *transfinite interpolation*, the method performs the mapping transformation between the natural and physical domain by interpolating, with *blending functions*, the curves that define the physical boundaries [34],[17]. This transformation can use more complex schemes whenever smoother meshes with good control of the aspect ratio of their elements are desired. For this purpose, *elliptic generators* are used [7],[9]. In general, these approaches achieve the final mesh by applying an iterative routine to an existing algebraic mesh.

Unstructured mesh generators have also been used to automatically construct quadrilateral meshes. These methods require more complex algorithms to be implemented; however, they lend themselves to mesh general geometries. *Grid superposition*, also known as *quadtree*, has been used to generate all-quadrilateral meshes. It consists of overlaying a uniform grid of points over the entire domain and properly connecting them to generate the mesh [26],[35]. Transformation from triangles is itself another method. Roughly speaking, the method combines two or more triangles and/or subdivides them to obtain quadrilateral elements.

Another technique used in conjunction with other schemes to generate quadrilateral meshes over complex geometries has been published under the title of *geometric decomposition*. This method subdivides the domain into simply connected polygons to which another method is applied to create the final mesh [36],[18],[19],[37].

Presented in this review as one of the most widely used unstructured methods, *advancing front* has been published as automatic triangular mesh generator capable of meshing complex geometries. If the advance of the front is associated with a transformation-from-triangles scheme, a fully quadrilateral mesh generation method is obtained [21]. Recently, a method called *paving* was developed by Blacker *et al.* to generate quadrilateral elements directly in the front [27],[28],[29],[30]. Published as an automatic mesh generator for plane surfaces, the method is discussed in this chapter and extended to mesh the surfaces of any general AutoCAD solid.

2.1 Mapping transformation

The mapping transformation was one of the first methods used in automatic quadrilateral mesh generation. The research and development that have been invested in this method produced a great number of versions widely used in the currently available finite element commercial packages. In this section, we present two versions of the method, namely *transfinite interpolation* and *elliptic generators*. Mapping techniques are relatively easy to implement; however, complex configurations depend on geometric decomposition and/or topological representation techniques in order to generate good meshes.

2.1.1 Transfinite interpolation

The basic scheme of this method uses linear *blending functions* to map a natural domain into the physical domain. The natural domain is represented as a square region with the natural coordinates (ξ, η) varying from zero to one. Each side of the natural domain is mapped into four parametric curves that enclose the physical domain [38],[39],[40]. Hence, any point $\mathbf{x} = (x, y, z)$ inside the physical domain can be obtained as

$$\begin{aligned} \mathbf{x} &= (1-\xi)f_1(\eta) + \eta f_2(\xi) + \xi f_3(\eta) + (1-\eta)f_4(\xi) \\ &\quad -(1-\xi)\eta\mathbf{x}_{12} - \xi\eta\mathbf{x}_{23} - \xi(1-\eta)\mathbf{x}_{34} - (1-\xi)(1-\eta)\mathbf{x}_{41} \end{aligned} \tag{2.1}$$

where $\mathbf{x}_{ij}$ are the four nodes defined by the intersection of the four parametric curves $f_1(\eta), f_2(\xi), f_3(\eta), f_4(\xi)$, defined as

$$f_i(t) = \big(x(t), y(t), z(t)\big) \qquad t = \eta, \xi \tag{2.2}$$

By inspection of equation 2.1 one can easily conclude that *blending functions*, different from linear, can be used to control the aspect ratio of the elements in the mesh.

A similar approach is adopted in *isoparametric interpolation* where only a few points of the boundaries are used [15]. This approach can also be used as a smoothing scheme to improve the mesh iteratively, in which the new position of a node is obtained as the average of its adjacent nodes [30].

2.1.2 Elliptic generators

This method generates the mesh by solving an elliptic differential equation that describes the transformation between the natural and physical domain [9],[38],[40]. Typically, the Laplace equation written in the physical domain governs this transformation:

$$\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}+\frac{\partial^2\phi}{\partial z^2}=0 \tag{2.3}$$

subject to $\phi = \xi,\ \eta$. This method is also known as *Winslow* or homogeneous *Thompson-Thames-Mastin* (TTM) generator [40].

Equation 2.3 is solved in the natural domain. The transformation yields

$$g_{22}\frac{\partial^2\psi}{\partial\xi^2}-2g_{12}\frac{\partial^2\psi}{\partial\xi\partial\eta}+g_{11}\frac{\partial^2\psi}{\partial\eta^2}=0 \tag{2.4}$$

subject to $\psi = x,\ y,\ z$, and

$$g_{11}=\frac{\partial\mathbf{x}}{\partial\xi}\cdot\frac{\partial\mathbf{x}}{\partial\xi}=\left(\frac{\partial x}{\partial\xi}\right)^2+\left(\frac{\partial y}{\partial\xi}\right)^2+\left(\frac{\partial z}{\partial\xi}\right)^2 \tag{2.5}$$

$$g_{12}=\frac{\partial\mathbf{x}}{\partial\xi}\cdot\frac{\partial\mathbf{x}}{\partial\eta}=\frac{\partial x}{\partial\xi}\frac{\partial x}{\partial\eta}+\frac{\partial y}{\partial\xi}\frac{\partial y}{\partial\eta}+\frac{\partial z}{\partial\xi}\frac{\partial z}{\partial\eta} \tag{2.6}$$

$$g_{22}=\frac{\partial\mathbf{x}}{\partial\eta}\cdot\frac{\partial\mathbf{x}}{\partial\eta}=\left(\frac{\partial x}{\partial\eta}\right)^2+\left(\frac{\partial y}{\partial\eta}\right)^2+\left(\frac{\partial z}{\partial\eta}\right)^2 \tag{2.7}$$

which are the components of the covariant metric tensor of the transformation. The computational stencil, using a second-order centered finite difference scheme for the numerical approximation of the first and second derivatives, is

$$\begin{aligned}\psi_{ij} \;=\; & C\Big[\frac{g_{22}}{\Delta\xi^2}(\psi_{i+1,j}+\psi_{i-1,j})-\frac{g_{12}}{2\Delta\xi\Delta\eta}(\psi_{i+1,j+1}-\psi_{i-1,j+1}-\psi_{i+1,j-1}+\psi_{i-1,j-1})\\ & +\frac{g_{11}}{\Delta\eta^2}(\psi_{i,j+1}+\psi_{i,j-1})\Big]\end{aligned} \tag{2.8}$$

where

$$C=\frac{1}{2\left(\frac{g_{22}}{\Delta\xi^2}+\frac{g_{11}}{\Delta\eta^2}\right)}\ . \tag{2.9}$$

Frequently, the final mesh is achieved by iteratively applying this stencil to an initial algebraic mesh, in which case the method is used as a smoothing scheme. The final mesh displays a high level of orthogonality throughout the entire domain, but tends to concentrate nodes around convex sectors of the boundaries and move them away from concave sectors [38]. These properties can be noticed in Figure 2-1*b*.

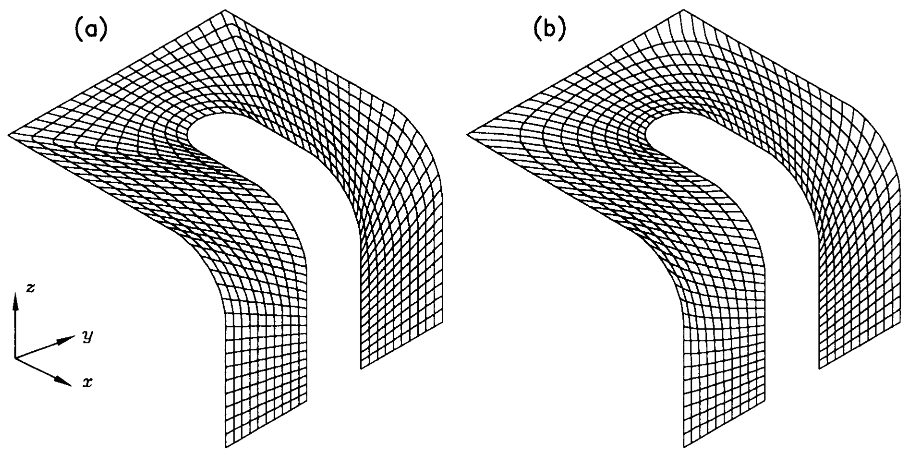

Figure 2-1: (*a*) Mesh generated with the transfinite interpolation scheme of equation 2.1; (*b*) the same problem after the mesh has been smoothed using the Laplace scheme of equation 2.8 and eighty iterations.

The Poisson equation scheme provides an equal level of orthogonality with better control of the mesh along convex and concave sectors of the boundaries [9],[38]. The governing equation is obtained by writing a Laplace nonhomogeneous equation:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2} = Q_\phi \tag{2.10}$$

also subject to $\phi = \xi, \eta$. Q_ϕ are the weight functions that provide mesh control. This equation is also solved using

$$g_{22}\frac{\partial^2 \psi}{\partial \xi^2} - 2g_{12}\frac{\partial^2 \psi}{\partial \xi \partial \eta} + g_{11}\frac{\partial^2 \psi}{\partial \eta^2} = -\, g\left(Q_\xi \frac{\partial \psi}{\partial \xi} + Q_\eta \frac{\partial \psi}{\partial \eta}\right) \tag{2.11}$$

where

$$g = \left|\frac{\partial \mathbf{x}}{\partial \xi} \times \frac{\partial \mathbf{x}}{\partial \eta}\right|^2 \tag{2.12}$$

2.2 Grid superposition

The first step in the grid superposition method is to overlay an orthogonal grid of points over the entire domain and connect them to form an initial mesh. Then, the points lying outside the boundaries are eliminated, and the remaining nodal points operate as the core of the whole mesh. Note that the mesh constructed so far has some triangular elements necessary to maximize the area meshed. In general, grid superposition requires some transformation of triangles in order to achieve a fully quadrilateral mesh. In the third stage, the initial mesh is connected to the boundary nodes using quadrilateral and triangular elements. Finally, this mixed mesh is transformed into an all-quadrilateral mesh.

Grid superposition has also been published under the title of *quadtree*. This name refers to the technique used to store the nodes of the overlaid mesh according to their spatial position. Tezuka in [41] combined the method with an advancing front scheme to complete the mesh along the boundaries and used an *a posteriori* approach of subdividing the triangular elements to obtain quadrilateral elements.

Baehmann *et al.* in [42] proposed a modified quadtree technique that divides the three-, four- and five-sided polygons created in the initial mesh into quadrilateral elements. The possibility of creating three types of polygons results in good control of the mesh density.

2.3 Geometric decomposition

Published as an automatic mesh generator, this method is, in reality, an auxiliary scheme that divides the domain into simple polygons which are then meshed using one of the methods discussed in this chapter. In planar configurations, the medial axis technique has been used to divide complex geometries. It consists of a set of interconnected curves containing the center of all circles that can be inscribed in the geometry. It provides the basis for the final division of the domain. Tam and Armstrong [18] and Krishnamoorthy *et al.* [37] used the medial axis technique along with other techniques to generate the final decomposition. Blacker *et al.* use the medial axis technique as an aid to form an abstract representation of the geometry [19]. The medial axis technique forms a sketch graphic representation of the domain indicating which pieces seem to project out from the geometry and must be decomposed first. Cheng *et al.* extended the decomposition up to the level of the final discretization, creating meshes with high-density gradients [43].

Souza and Gattas proposed a scheme that represents the surface to be meshed with meshed patches [20]. The first step consists of overlaying an initial set of regular meshed patches. Then, the intersections between the boundaries and the patches are determined. In a third stage, the parts of the initial regular meshed patches that lay outside the surface are deleted. Finally, the patches are connected to the boundaries in order to form the mesh. Note that each one of the regular meshed patches represents a division of the entire domain.

2.4 Transformation from triangular meshes

Any triangle can be divided into three quadrilaterals. This fact opens the possibility of using any existing triangular mesh generation method to create a quadrilateral mesh [25]. Triangles can also be merged to produce quadrilateral elements [26]. The former scheme generates a mesh with finer density if compared to the initial triangular mesh, whereas the latter one ends up with a coarser density. However, the combination of both is often necessary to guarantee a quadrilateral mesh with reasonable element aspect ratios. Johnston *et al.* have developed an algorithm that attempts to generate good quadrilateral elements whose quality is associated with the quality of the initial triangular mesh [23]. Rank *et al.* proposed a scheme to preserve the original density distribution during the transformation [24]. Basically, the scheme combines two triangles to form a quadrilateral that is then divided into four quadrilaterals. The mesh density obtained is closer to the original density than it would be if the two triangles had been directly divided into six quadrilaterals.

The above approaches require a relatively simple code, taking advantage of the speed and robustness of the existing triangular mesh generators capable of discretizing very complex geometries.

2.5 Advancing front

The advancing front method has been successfully used to automatically generate triangular meshes over general geometries. The robustness of the technique is based on the fact that any polygon can be decomposed into triangles, so that the closure of the mesh can always be achieved. Recently, some research has been conducted to generate quadrilateral elements with similar approaches. Zhu *et al.* proposed a method that advances the front by creating and combining triangles, still in the front, to generate quadrilateral elements [21]. This algorithm requires a simply connected domain; i.e., all internal boundaries of the domain must be connected to its external boundary using *cut lines*.

Another method developed to directly generate quadrilateral elements in the front was introduced by Blacker *et al.* under the title of *paving* [27],[28],[29],[30]. This method advances the front by projecting rows of quadrilaterals inward, so that the elements near the contour tend to have a good aspect ratio, contributing to the orthogonality of the mesh along the boundaries (i.e., the perpendicularity of the lines of the mesh). In this method, as in the work published by Zhu *et al.*, the closure is guaranteed if the front has an even number of nodes, which is achieved by maintaining this condition throughout the generation. Note that a front with five nodes, for instance, can be closed only with at least one triangle.

Paving has been used to automatically generate quadrilateral meshes over general planar geometries. The complexity of the algorithm, however, leads to a relatively low speed of generation if compared to the advancing front method used to produce triangular meshes.

As already pointed out, the present research extends *paving* to mesh general curved surfaces. We also attempt to improve the speed of generation by introducing a circular advance of the front.

Chapter 3

Implementation of an Automatic Quadrilateral Mesh Generator for General Surfaces

The combination of triangles has been proposed as a method to generate quadrilateral meshes, in which the initial triangular mesh is created using well-published techniques, such as advancing front and grid superposition [21],[22],[23],[24],[25],[26],[35]. This method, however, fails to preserve the mesh orthogonality (i.e., the perpendicularity of the lines of the mesh) and low element distortion (i.e., nearly square elements) along the boundaries which are strongly desired properties.

Structured mesh generators have been extensively used to create quadrilateral elements in finite element packages. These methods are, basically, transfinite mappings and require that the geometry has a rectangular-like shape [34],[7],[8],[17]. The mesh generated is geometrically pleasing, which means that boundary orthogonality and low element distortion are normally preserved, especially when the algebraic generation is followed by an elliptic smoothing scheme.

The advancing front method has also played an important role in the generation of meshes in applications involving complex geometric contours. However, only recently a scheme that creates quadrilaterals directly, without combining triangles, was published. Known as *paving*, this technique produces meshes with low element distortion along the boundaries and lends itself to meshing any general planar geometry [27],[28],[29],[30]. In this thesis, we modify the original algorithm of *paving* to mesh any kind of surface. This implementation requires that the algorithm keep track of the surface with frequent evaluation of its normal vector and projection of any new node created or modified. It also demands the analytical description of the surface that must be extracted from the solid to be meshed. To solve these difficulties we use the AutoCAD Development System (ADS) with the Application Programming Interface library (API) that allows the use of the C-programming language along with the drawing capabilities of AutoCAD, version 12. This approach of integrating a CAD package and the pre-processing of the finite element analysis has been proposed by Rolph III [10], Watson *et al.* [44], and Jiazhen [45].

The method presented in this text starts the generation in the boundaries and projects rows of elements inward as proposed in *paving*; however, some modifications are introduced due to the necessity of generating the mesh on curved surfaces. For instance, the check of intersections performed during the advance of the fronts has to be done *a priori*, i.e., before generating a new row of elements (see Section 3.8). Note that the check of intersection of lines in a curved surface is computationally expensive.

3.1 Geometry extraction: integration with CAD

This part of the algorithm provides a powerful tool for automating the mesh generation which reduces the time consumed and the possibility of error in the pre-processing stage of any finite element analysis.

First, we must establish some notation that is referred to hereafter. Following Figure 3-1, we have

- *surface*: it delimits the solid volume. It is bounded by the edges of the solid and has the same shape of the primitive entity that has created it (plane, cylinder, cone, etc...).

- *edge*: it is created by the intersection of two surfaces of the solid.
- *loop of edges*: a closed path of edges. A surface is bounded by one or more loops of edges. Note that a loop of edges may contain only one edge as in the case of the border of a hole (see Figure 3-1).

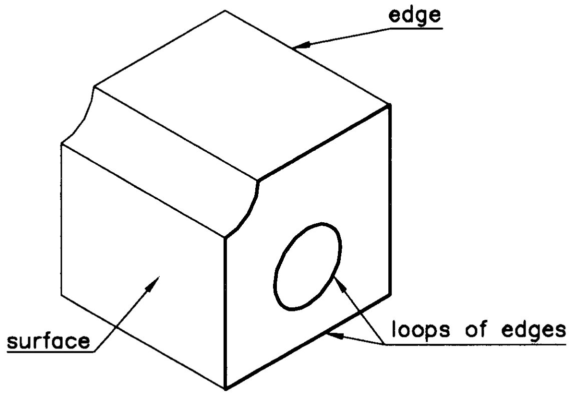

Figure 3-1: Solid created in AutoCAD and notation adopted.

The geometry extraction consists of constructing, for each surface of the solid, a database of nodal points equally spaced along the edges of the solid in accordance with the mesh density requested by the user. This database is then directly transformed into initial fronts, which are the starting point for the mesh generation.

AutoCAD organizes the geometric parameters of the solid in an object-oriented structure so that each surface has its own identification number stored in a linked list. This list is accessed via ADS-API functions. In the C-code, this data is stored in two arrays of structure which are constructed to process the geometry extraction: the first array contains the identification numbers of the edges of the solid, their lengths and their parametric descriptions that include the starting and ending points of the edge and the corresponding values of the parameters for these points; the second array contains the identification numbers of the surfaces, their types (plane, cylinder, cone, etc...) and the identification numbers of all the edges that bound each surface of the solid. These two arrays contain the basic information needed from AutoCAD in order to form the fronts of each surface following the steps below.

1. Generation of the nodes in the edges of the solid.
2. Setting up of the initial front: the nodes created in the edges are arranged in a closed loop to form the starting point of the mesh generation.
3. Establishment of the front orientation: the direction in which the nodes of the front are oriented in the loop (clockwise or counterclockwise).
4. Evaluation of angles and distances in the front: the algorithm calculates the angles between adjacent segments of the front and the distances between nodes. Note that a segment of the front is defined as the line joining two adjacent nodes.

Note that the explicit indication of the type of the array (array of structure) was included in the text to stress the object-oriented nature of the code developed to test the algorithm described in this chapter. Note also that a variable type structure is a general feature of the C-language used for the code (see Chapter 4).

3.1.1 Generation of nodes on the boundaries

The general mesh density is one of the inputs required for the algorithm. Given as the size of a regular quadrilateral, the density is initially used to equally divide each edge into a sequence of interconnected nodes. During this division, the condition of an even number of nodes is imposed, so that each initial front of a surface will also have an even number of nodes. The approach of dividing each edge individually is necessary to match the compatibility between two adjacent surfaces of the solid. Different densities are also possible. It requires user interaction to select the edges of the solid and the local density values attributed to them.

3.1.2 Local density control

The mesh density initially selected is applied in all edges of the solid, creating sequences of nodes equally spaced. It generates a uniform mesh over the surface of the whole solid. However, different mesh densities are required in almost every finite element analysis, either to capture the numerical solution more accurately or to reduce the computational effort by using a coarse mesh in regions where the solution is smooth.

The input necessary for the method is a set of closed loops of nodes distributed along the boundaries, which reflects the density desired for the final mesh. Therefore, the local mesh density is implemented by selecting edges with different element sizes as depicted in Figure 3-2.

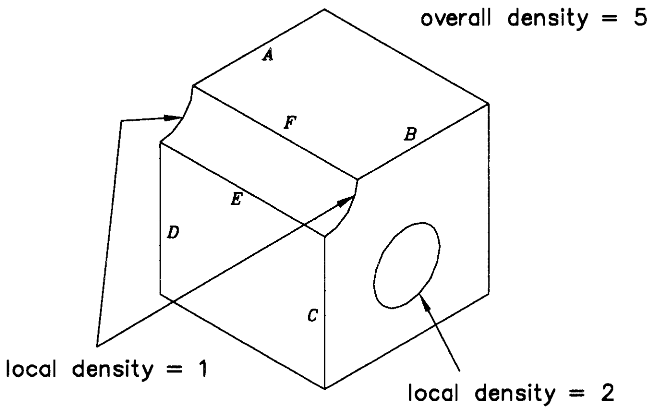

Figure 3-2: Local density input.

The method supports density variation along the front; however, abrupt variations of the element size along the front can create a highly distorted mesh and even crash the algorithm. Hence, an arithmetic ratio is imposed on the distances between the nodes of the edges adjacent to those where different densities have been selected. In Figure 3-2, the edges A, B, C, and D are submitted to this process in order to smoothly migrate from the local density of '1' to the overall density of '5'. As an example, let us take the length of the edge B equal to '18'. The number of divisions is given by the formula used for summing up the terms of an arithmetic series:

$$L = \frac{k}{2}\,(a_0 + a_n) = 18 \tag{3.1}$$

where $a_0 = 1$ and $a_n = 5$ are the first and last element of the series and k is the number of divisions we want to calculate. In this example, we took a length that provides an exact number of divisions ($k = 6$), which does not occur in most cases. Therefore, the algorithm takes the even integer closer to the value of k calculated as above.

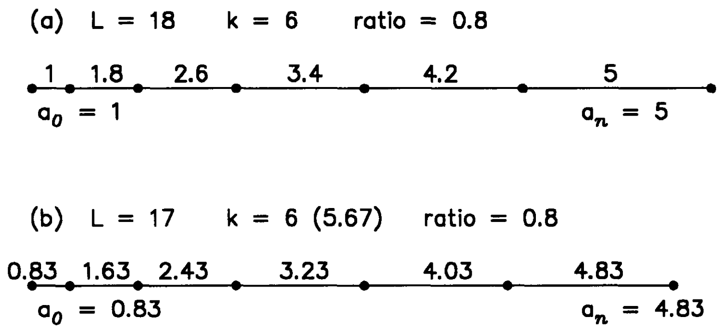

Figure 3-3: Example of smooth density variation along the edges of the solid.

The ratio is obtained also as in an arithmetic series:

$$\textit{ratio} = \frac{a_n - a_0}{k-1} = 0.8 \tag{3.2}$$

This ratio represents the increment of the interval between nodes as depicted in Figure 3-3*a*. In Figure 3-3*b*, we illustrate a case where $k = 5.67$ and the algorithm takes $k = 6$ to calculate the $\textit{ratio} = 0.8$ which, in this case, sums up a total length of 18 units of drawing. To adjust it to the correct length (17), we subtract $1/k$ of the difference in length (1/6) from the first element of the sequence. Note that it is similar to subtract 1/6 from each element of the arithmetic sequence.

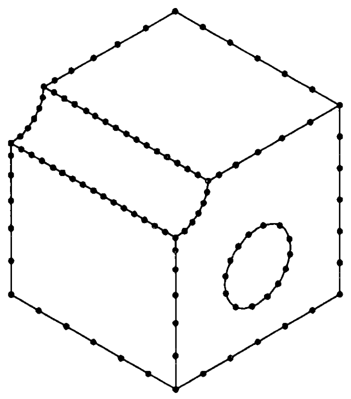

Figure 3-4: Generation of nodes in the edges of the solid with local density implemented.

The result of the node generation along the boundaries, with local mesh density selected as in Figure 3-2, is illustrated in Figure 3-4. Note that the edges E and F end up with a uniform density equal to '1'.

3.1.3 Implementation of the density distribution in the mesh

After establishing the density on each edge, the method advances the front and uses the correction of the element sizes in the front (see Section 3.4) to migrate from the different densities to the average density calculated for each surface of the solid. This is done to allow a smooth transient between two fronts as, for instance, when a surface with a hole is meshed.

The use of a density distribution function is possible since the routine that corrects the element sizes of the front can compare the local size of the mesh with any given value. It does not introduce any substantial alteration in the method itself, but requires constant checks of the spatial position along the front in order to obtain the values to which the element sizes have to be adjusted.

3.1.4 Initial fronts

The initial fronts of a surface are constructed by connecting the sequence of nodes created in each edge of the solid as described in Subsections 3.1.1 and 3.1.2. The first step is the generation of the loops of edges as illustrated in Figure 3-1. The connectivities of the edges that bound each surface are established by comparing the ending points of the edges of the solid. This chain of edges is associated with the nodes generated in Subsections 3.1.1 and 3.1.2, and a unified sequence of nodes is obtained and stored in an array of structure (see Figure 3-5).

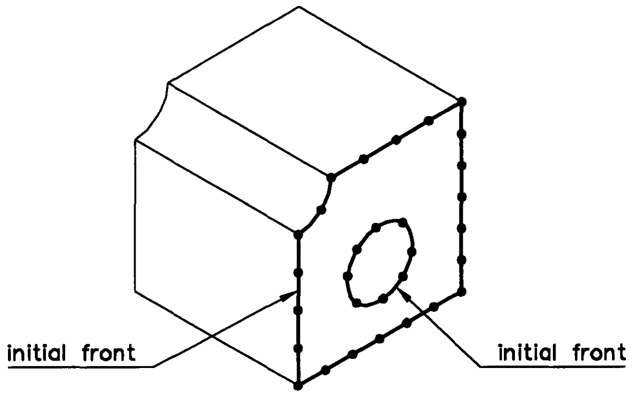

Figure 3-5: Initial fronts of a surface of the solid.

The status label, f, is an integer variable created to delimit the various fronts of a surface and controls the shrinking and expansion of each front size during its advance. It also regulates the creation of new fronts and their closures.

Figure 3-6 illustrates the delimitation of the fronts of a surface with the status label. The label '0' indicates the beginning and the end of the first front, whereas the label '-1' indicates the sequence of nodes within the two extreme nodes of the front labeled '0'. The same occurs for the label '1'. In this case, we have a front with the nodes n = 20, 17, 12, 13, 24, 19, 21, 14. The last example shown in Figure 3-6 displays a front already closed (label '2'), in which there are no '-1' labels.

n →	2	4	5	0	10	1	8	3	6	9	13	7	15	18	16	11	20	17	12	13	24	19	21	14	22	30	27	34	25	26
f →	0	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	0	1	−1	−1	−1	−1	−1	−1	1	2	2	2	2	2	2

Figure 3-6: The use of the status label to control the fronts of a surface.

3.1.5 Orientation of the fronts

The front orientation is necessary to calculate the angle at each node of the front. Since we are not only considering the particular case of meshing a plane, it is mandatory to use the vector normal to the surface along with the front orientation for the evaluation of angles (see Subsection 3.1.6).

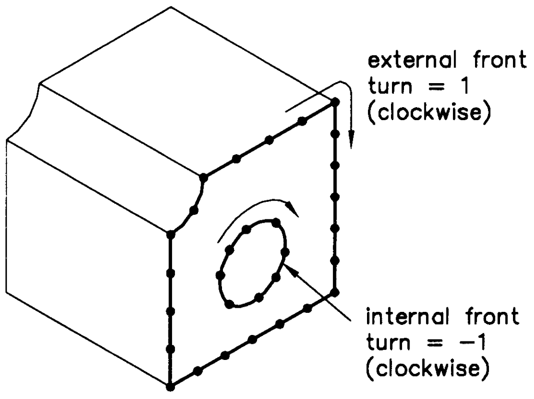

Figure 3-7: Orientation of the initial front.

We use the global variable *turn* (see Appendix A.2) to control the orientation of the front. The value '1' is assigned to *turn* if the front is oriented clockwise and the value '-1' is assigned to *turn* otherwise. Note we consider an observer located outside the solid (see Figure 3-7). This convention is valid only for external initial fronts; i.e., fronts that are not the border of holes. For internal initial fronts, $turn = 1$ means counterclockwise and $turn = -1$ means clockwise.

Following the scheme depicted in Figure 3-8, the orientation of the front is evaluated using the vector normal to the surface, $\underline{n}$, at the first node of the front (node i). Initially, the vector $\underline{t} = \underline{n} \times \underline{v}$ is calculated. Then, the point p is projected onto the surface in order to check whether it lies inside the boundaries of the surface. If it lies outside, as shown in the figure, the value '-1' is assigned to the variable *turn*.

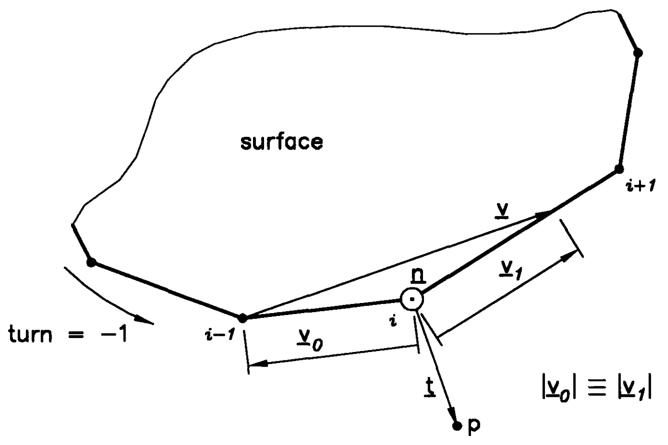

Figure 3-8: Scheme to determine the front orientation.

Note the way of turning associated with the front orientation (clockwise and counterclockwise) is an abstraction to help understand the concept. It is possible to have a front composed of external and internal sectors. In such a case, the front turns in two opposite directions but the variable *turn* has only one value.

3.1.6 Evaluation of angles and distances in the front

The algorithm evaluates the angle at each node of the front and the distance between two adjacent nodes following the scheme illustrated in Figure 3-9. First, the vector

$\underline{t} = \underline{v} \times \underline{n}$ is calculated. Then, the vector $\underline{t}$ is multiplied by the value of the variable *turn* (-1 in this case), so that the result always points inward; i.e., in the direction that the front advances. Note that $\underline{n}$ is the vector normal to the surface at the node i. The semi-angle, a, is calculated using the inner product between one of the two vectors $\underline{v_0}$ or $\underline{v_1}$ and the vector $\underline{t}$. The use of the semi-angles is necessary since the value returned by the C-function $acos(x)$ varies from 0.0 to π while the angles can assume any value up to 2π. The angle at the node i is defined as $2a$.

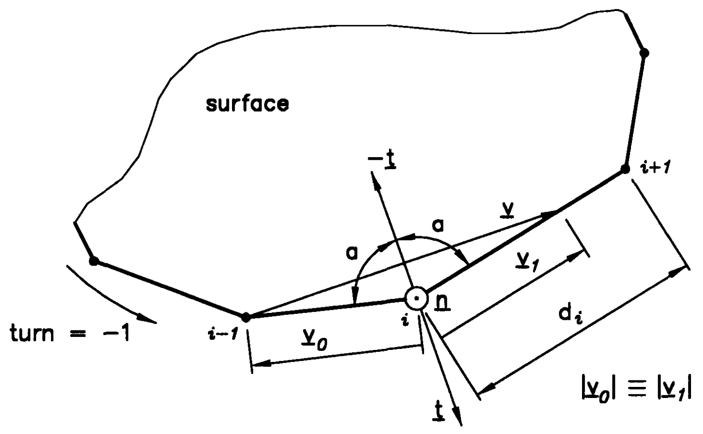

Figure 3-9: Angle, $2a$, and distance, d_i, of a front node.

The distance between nodes, d_i, is calculated and stored in the front database as part of the first node following the orientation of the front. Note the subscript i of the distance d_i in Figure 3-9 represents this dependence.

The angle and distance are calculated for every node in the initial front as well as every time a local alteration in the front is executed to advance the front.

3.2 General structure of the algorithm

Once the geometry has been extracted and the initial fronts of each surface of the solid created, the mesh generation can be executed. Roughly speaking, the main structure of the algorithm consists of a loop that in each sequence checks all the fronts of a surface to assure that one of the fronts can advance without intersections. After advancing the front, another check to correct the element sizes along the new front is carried out before starting the next sequence of the loop.

The checks executed before advancing the front search for the conditions required to close the front, seam nodes, close corners, or intersect fronts. If one of these checks is positive, the corresponding modification is performed in the front. In the case of a positive closure or intersection check, the sequence of the loop is broken after the execution of the correspondent modification; i.e., a new sequence is started without advancing the front. The complete structure of the algorithm is represented below.

```
Advance all initial fronts
REPEAT for all fronts
     IF check of Closure is positive THEN break the loop
     REPEAT while Seaming of nodes is positive
          IF check of Closure is positive THEN break the loop
     END of loop
     REPEAT while Closing Corner is positive
          IF check of Closure is positive THEN break the loop
     END of loop
     IF Check for intersection is positive THEN break the loop
     IF Smoothing of the front is positive THEN
          IF Check for intersection is positive THEN break the loop
     Check for intersection
     Advance the front
     IF Correct the element sizes in the front is positive THEN
          Smooth the front
END of loop
Smooth the mesh
Clean up the mesh
```

The seaming-of-nodes check is performed as part of another loop that is not ended until a negative check is detected. Inside this loop, a check of closure is executed, since every time two nodes are seamed the number of nodes in the front decreases. If no closure is detected, the sequence of the loop continues and a similar loop is executed to check the closing of corners. Again, if no closure is observed a check for intersections is carried out to verify all fronts against all fronts including each front against itself. The last routine executed, before the advance of the front is allowed, is the smoothing of the front. If the front needs to be smoothed, another check of intersections is performed. After completing the mesh, a smoothing routine is executed to improve the aspect of the elements. Finally, a cleaning up routine is called to correct the occurrence of nodes connected to only two elements.

As displayed above, the algorithm checks intersections before advancing the front and not after having performed this function as proposed in *paving* for planar surfaces [28],[30]. This approach was adopted to allow the generation over any surface, planar or not. In the case of a plane, it is relatively easy and fast to check the intersection of lines; however, it becomes a time-consuming process when a curved surface is considered.

The order in which each one of the checks and routines is executed is quite important to advance the front and continue the mesh generation. Some modifications, however, seem to introduce distortions in the mesh rather than solve them. Actually, a few distorted elements are created following rules that allow the next checks and/or routines to correct them. Ultimately, these distorted elements are solved in the smoothing of the final mesh and in the cleaning up routine. We illustrate this interrelationship among the different checks and routines executed in the algorithm with two examples depicted in Figure 3-10. In Figure 3-10*a*, the sharp angle is initially solved by seaming two nodes, which creates a new node connected to only two elements. These two distorted elements are finally corrected with the cleaning up routine. The other example, depicted in Figure 3-10*b*, adopts a similar sequence to eliminate distortion. The last step of the correction, however, is achieved with the smoothing of the final mesh.

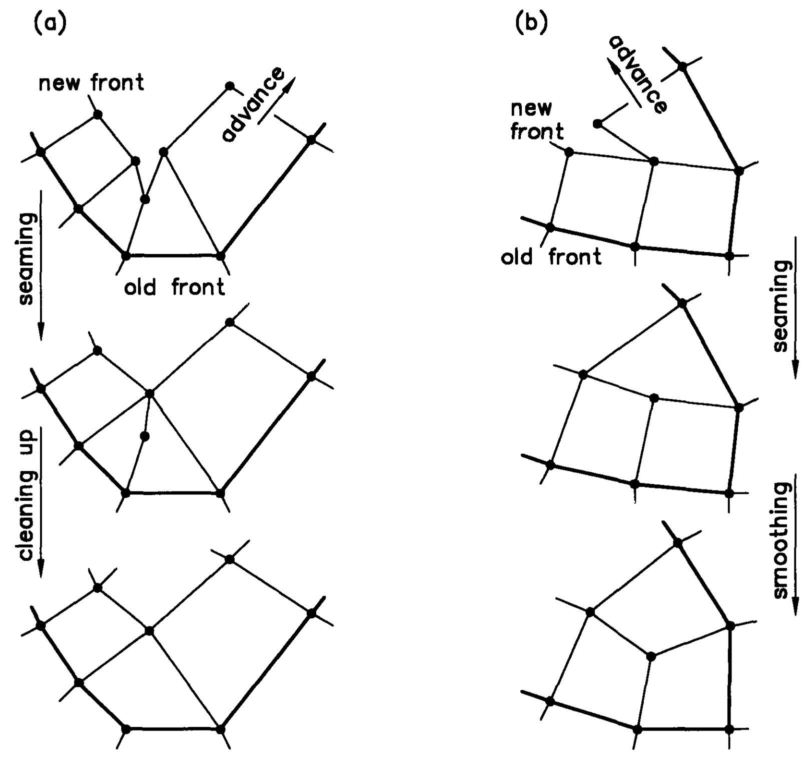

Figure 3-10: Two examples of sequences executed to enhance the quality of distorted elements. Note that in (*a*) we do not represent the smoothing of the final mesh, which is executed before the cleaning up routine.

Here, we stress the central part that the smoothing routine plays in the correction of distortions. Literally, any element distortion, not solved by the checks and routines executed during the advance of the front, is corrected in the final smoothing of the mesh. It excludes, of course, the one treated in the cleaning up routine, which is carried out after the smoothing.

3.3 Circular advance of the front

The front is advanced by creating a closed loop of elements along the front. The sequence is constructed by placing a new element after the last one created. Before placing a new element, the function checks the angle at the next node of the front in order to select the routine that shall create the element. This process is performed every time a new element is generated attempting to reduce element distortion and obtain mesh orthogonality especially along the boundaries of each surface.

3.3.1 First element of the front

The initial step to advance the front is the creation of the first element connected to two adjacent nodes with angles between 135 and 240 degrees. The points p_1 and p_2 shown in Figure 3-11 are created using the same vector $-\underline{t}$ displayed in Figure 3-9 with magnitude $h_i = (d_{i-1} + d_i)/2$ and $h_{i+1} = (d_i + d_{i+1})/2$, respectively. These points, after being projected onto the surface, complete the four nodes needed for the first element.

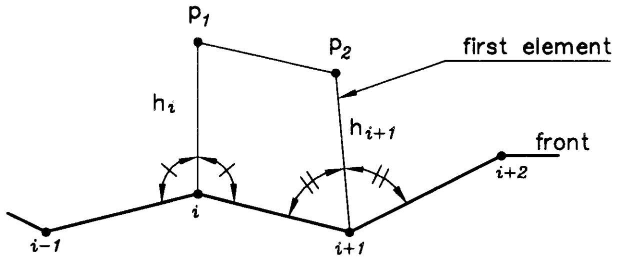

Figure 3-11: Generation of the first element of the front.

As the front approaches the closure, it is possible that the method did not find any pair of adjacent nodes with angles in the range required to create the first element. For this case, the algorithm divides the front as illustrated in Figure 3-12. These two fronts are treated as new fronts in the algorithm, so that the seaming of nodes, closing corner, closure, and check of intersections are carried out before advancing the front again. In general, these fronts are ended in the closure check.

With the first element constructed, a sequence of elements is created along the front line. The elements are generated in accordance with the angle of the front as discussed in the following subsections.

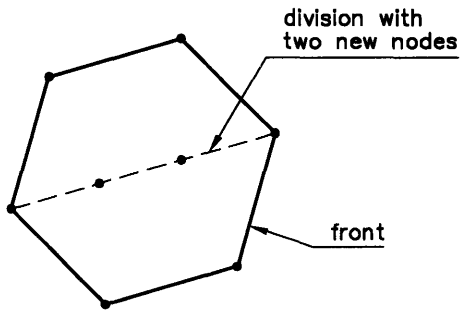

Figure 3-12: Division of a front in which the first element cannot be created.

3.3.2 Generation of an element at a corner

Nodes with angles equal to or less than 135 degrees are considered corners in the algorithm, so that no new point is necessary to complete the four nodes needed for the element. As illustrated in Figure 3-13*a*, the node subsequent to node i is the fourth node necessary to form the new element.

An additional condition is imposed on the generation of the element in order to reduce the distortion of the next element to be created in the front and correct the shape of the one already generated. If the angle at the node i is greater than 90 degrees, the shape of the new element is corrected by moving the node p as depicted

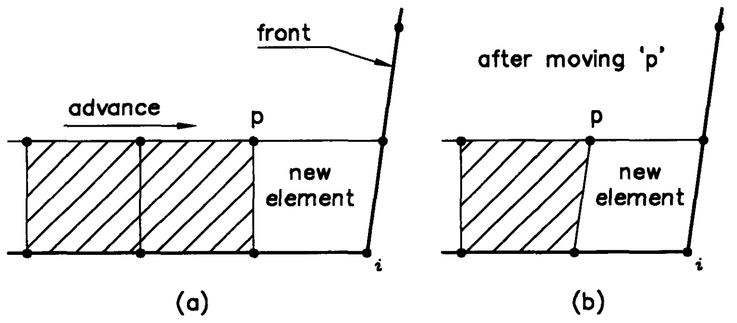

Figure 3-13: (*a*) Generation of an element at a corner; (*b*) correction of the element shape.

in Figure 3-13*b*. This action transforms the new element in a parallelogram. It is important to note that any distortion produced in this stage of the generation is eliminated in the final smoothing of the mesh as described in Section 3.10; however, the procedure of reducing distortion, right after the construction of each element, is necessary to avoid the cumulative effect it might have over the next elements created and, ultimately, over the entire mesh.

3.3.3 Generation of an element in a straight sector

Another possible case faced by the front during its advance is the generation of a new element in a straight sector of the front which is defined as a node with an angle greater than 135 degrees, and equal to or less than 240 degrees. In this case, the fourth node necessary to generate the new element must be created. The scheme is illustrated in Figure 3-14.

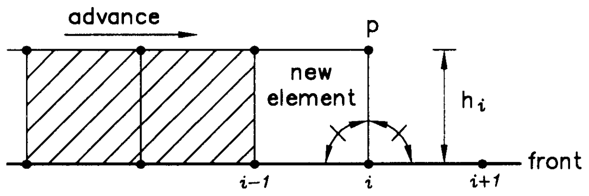

Figure 3-14: Generation of an element in a straight sector of the front (node i).

The algorithm creates the point p using the same vector $-\underline{t}$ displayed in Figure 3-9. This point is set at the distance $h_i = (d_{i-1} + d_i)/2$ from the node i and projected onto the surface to create the element.

Still in the case of a straight sector, the method checks the angles from node $i - 3$ to node $i + 3$, searching for values equal to or less than 60 degrees. In the event that this condition occurs, the length h_i is multiplied by a factor with magnitude less than

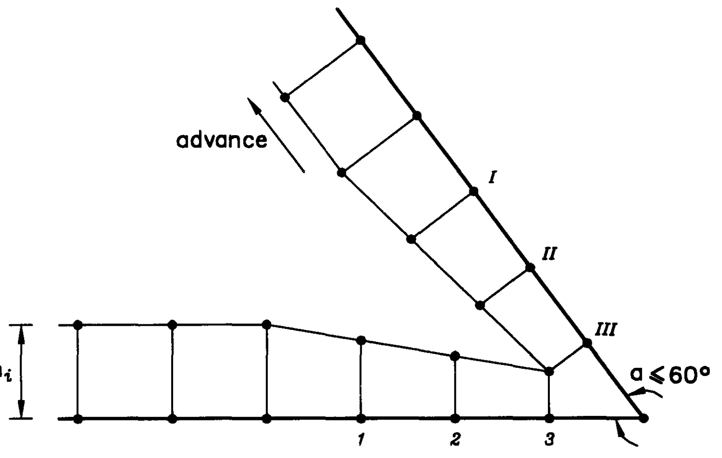

Figure 3-15: Result obtained with the reduction of the length h_i near a sharp corner of the front.

one, as illustrated in Figure 3-15:

$$h_1 = h_I = \frac{5}{8}\, h_i \tag{3.3}$$

$$h_2 = h_{II} = \frac{7}{8}\, h_i \tag{3.4}$$

$$h_3 = h_{III} = \frac{\sin a}{2}\, h_i \tag{3.5}$$

The reducing factor of the two nodes close to the corner is set proportional to the sine of the angle a in order to avoid the intersection of the new front with itself. Note that the small angle formed in the new front is solved with the seaming of nodes.

3.3.4 Generation of elements around a wedge

This case is solved by generating two elements in the front. As displayed in Figure 3-16, three new nodes are created around the wedge. The method considers as a wedge any node with an angle greater than 240 degrees, and equal to or less than 300 degrees.

The vector $\underline{v_0}$ is collinear to the segment delimited by the nodes i and $i+1$ and has norm $|\underline{v_0}| = (d_{i-1}+d_i)/2$. Similarly, the vector $\underline{v_1}$ is collinear to the segment delimited

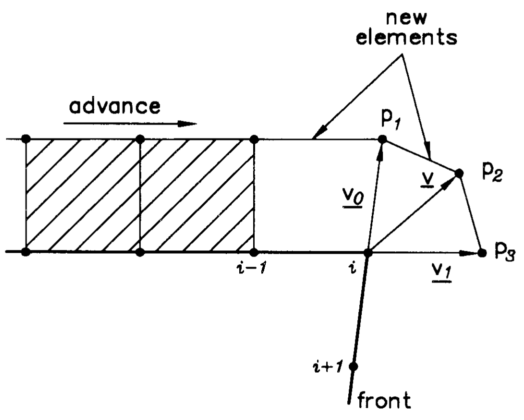

Figure 3-16: Generation of two elements around a wedge.

by the nodes $i-1$ and i, with $|\underline{v}_1| = |\underline{v}_0|$. The vector $\underline{v}$ is the resultant of the sum of the vectors $\underline{v}_0$ and $\underline{v}_1$, with norm $|\underline{v}| = |\underline{v}_0|$. This normalization is necessary to avoid overlaps not detected in the check of intersections. The distortion of the elements introduced with this normalization is partially solved in the smoothing of the front described in the Subsection 3.7. Note that the two new elements are created only after the projection of the nodes p_1, p_2, and p_3 onto the surface.

3.3.5 Generation of elements around a sharp wedge

This routine is similar to the one discussed in Subsection 3.3.4. The algorithm creates three elements to advance the front around the sharp wedge, which is defined as a node with an angle greater than 300 degrees.

As illustrated in Figure 3-17, the vector $\underline{v}'$ is created as

$$\underline{v}' = -\frac{d}{|\underline{v}|}\,\underline{v} \tag{3.6}$$

where

$$d = \frac{d_{i-1} + d_i}{2} \tag{3.7}$$

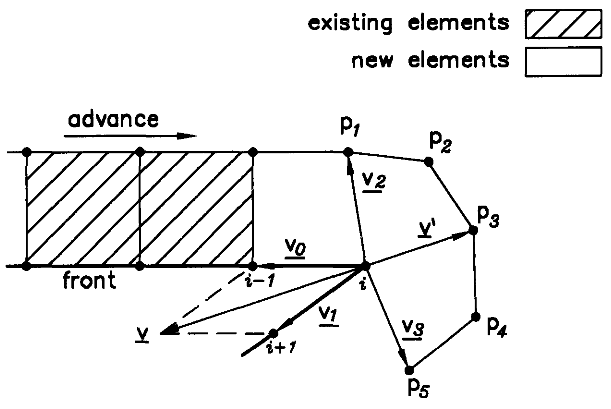

Figure 3-17: Generation of three elements around a sharp wedge.

Then, the two other vectors $\underline{v_2}$ and $\underline{v_3}$ are evaluated to set the points p_1, p_3, and p_5:

$$\underline{v_2} = \frac{d}{|\underline{v_0} + \underline{v'}|} (\underline{v_0} + \underline{v'}) \tag{3.8}$$

$$\underline{v_3} = \frac{d}{|\underline{v_1} + \underline{v'}|} (\underline{v_1} + \underline{v'}) \tag{3.9}$$

By adding the vectors $\underline{v_2}$ and $\underline{v_3}$ to vector $\underline{v'}$ and normalizing them accordingly, we obtain the other two points p_2 and p_4 which completes the number of nodes required to construct the three new elements. We also have to project these five points onto the surface before creating the elements.

As mentioned before, the normalization of these vectors is necessary to avoid intersections not covered in the check of intersections performed in the algorithm. The distortions introduced in the elements due to this normalization are partially solved in the smoothing of the front described in Section 3.7. The final correction is carried out with the smoothing of the mesh described in Section 3.10.

3.3.6 Last element of the front

The last element created in the front closes the sequence of elements generated to advance the front. The method considers the node p_1 of the first element (see Figure 3-11) as one of the nodes needed for the construction of the last element in the front. Note that the use of the node p_1 connects the last element of the sequence to its origin. Some configurations might require the construction of more than one element to complete the front. For instance, if the first element is located after a wedge, two new elements have to be created to complete the advance of the front. In this case, the generation of the last elements of the front is carried out using the same procedures described in the previous subsections.

After completing the advance of the front, the angles and distances of the new front are evaluated as described in Subsection 3.1.6.

3.4 Correction of the element sizes in the front

Executed immediately after advancing the front, the correction of the element sizes in the front attempts to maintain the average mesh density of the surface. Typically, the distance between nodes increases or decreases whenever the advance is carried out in a convex or concave sector of the front, respectively; however, other operations on the front can also create these situations.

As illustrated in Figure 3-18, the method checks the front distances and selects the nodes that shall be joined in order to increase the distances between nodes (Figure 3-18*a*) or be submitted to the insertion of a new element to reduce these distances (Figure 3-18*b*).

The criterion used to determine whether a sector of the front is eligible to be corrected or not is quite simple. If the values of the two distances between three adjacent nodes in the front (nodes $i-1$, i, and $i+1$) are less than d_{min}, the procedure for joining these nodes is executed. If the values of these two distances are greater

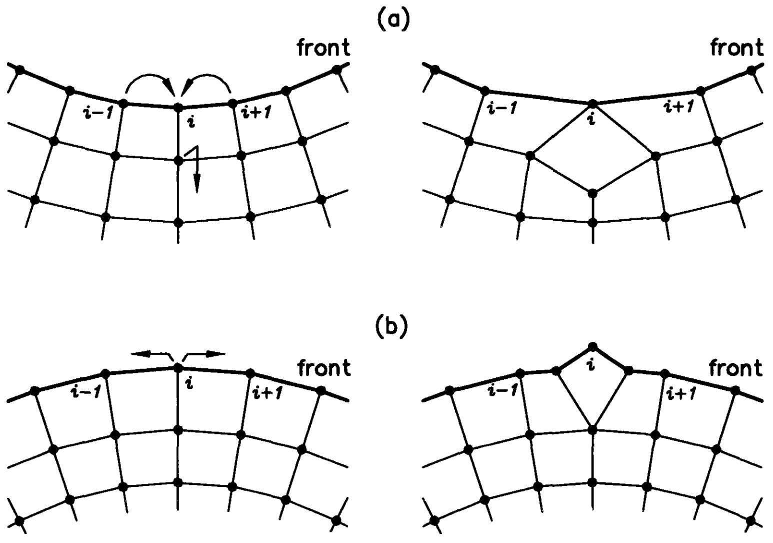

Figure 3-18: Correction of the element sizes in a concave and convex sector of the front.

than d_{max}, a new element is inserted:

$$d_{min} = 0.7\ d_{average} \tag{3.10}$$

$$d_{max} = 1.3\ d_{average} \tag{3.11}$$

$$d_{average} = \frac{1}{n} \sum_{k=1}^{n} d_k \tag{3.12}$$

where n is the number of nodes in the fronts of the surface.

The correction of the element sizes modifies the distances locally in the front. To complete the correction, a smoothing routine is executed along the front to reduce the difference in distance between adjacent nodes (see Subsection 3.7).

The correction of the element sizes in the front is also the key function used to manage the transition among different mesh densities. The use of local mesh densities, as discussed in Subsection 3.1.2, requires that the distance between nodes be adjusted as the front advances in order to manage the integration of the different densities. It is especially useful in cases where a different local density is requested for an edge of the solid not connected to any other boundary. Figure 3-4 illustrates this case.

It demands the migration of the various densities ('1', '2', and '5') to the average density of the surface, $d_{average}$. Note that, as in any modification of the mesh, the nodes created or moved with the correction of the element sizes have to be projected onto the curved surface.

3.5 Closing corner

The closing corner routine was introduced in the algorithm in order to solve one case that neither the intersection check nor the advance-of-the-front routine itself deals with. Attempting to generate uniform meshes over surfaces with rectangular-like shapes, the check of intersection does not consider nodes within the range $i-3$ to $i+3$ as candidates to intersect a given node i. This approach can be understood if we imagine that the bold line displayed in Figure 3-19*a* is the front of a mesh under construction over a rectangular surface. For a given uniform mesh density, in general, the element size in the two vertical edges of the surface differs slightly from the element size in the two horizontal edges of the same surface, due to the imposition of the condition of an even number of nodes in the initial front. It introduces a little distortion in the front as it advances far from the initial front. If we allow intersection between nodes i and $i+3$, the final mesh will not be uniform. That is, it will be different from the mesh that would be created if a transfinite interpolation method were used. Figure 3-19*b* shows the solution desired for the mesh, which will display a uniform pattern after being smoothed. Note the mesh illustrated in Figure 3-19*a* ends up in a nonuniform mesh. It is also important to point out that the definition of a rectangular surface embodies more than a simple planar parallelogram. A cylinder with a longitudinal cut has a rectangular curved surface along its lengthwise dimension.

The scheme performed in the closing corner routine is depicted in Figure 3-20. The first condition necessary to close the corner is the existence of two adjacent nodes with angles equal to or less than 135 degrees (nodes i and $i+1$). Once this condition is satisfied, the distance between the nodes $i-1$ and $i+2$ is compared with

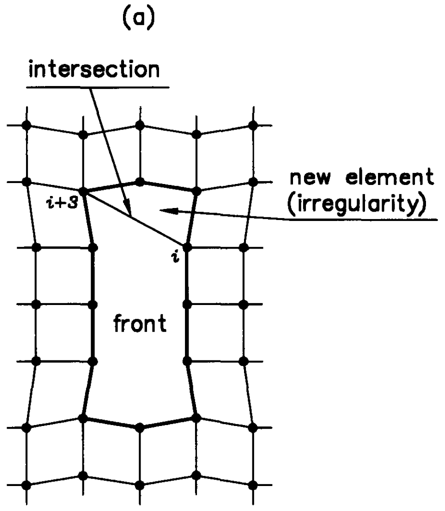

Figure 3-19: Irregularity that would be created in the mesh of a rectangular surface if the intersection between nodes i and $i+3$ were allowed.

the magnitude of the vectors used to advance the front in these nodes:

$$\mathrm{dist}(i-1,\ i+2)\ <\ 1.05\ (h_{i-1}+h_{i+2}) \tag{3.13}$$

This condition is necessary to avoid the generation of distorted elements with the sequential execution of the closing corner routine, as illustrated in Figure 3-21, whereas the condition of small angles is necessary to prevent the undesirable case represented

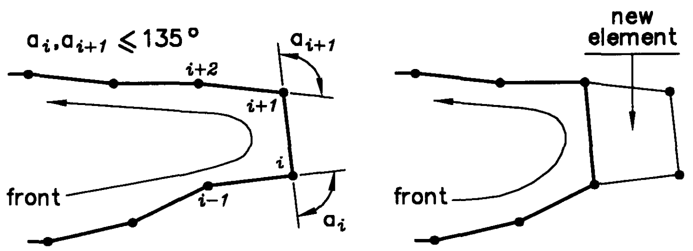

Figure 3-20: Scheme of the closing corner routine.

in Figure 3-19*a*.

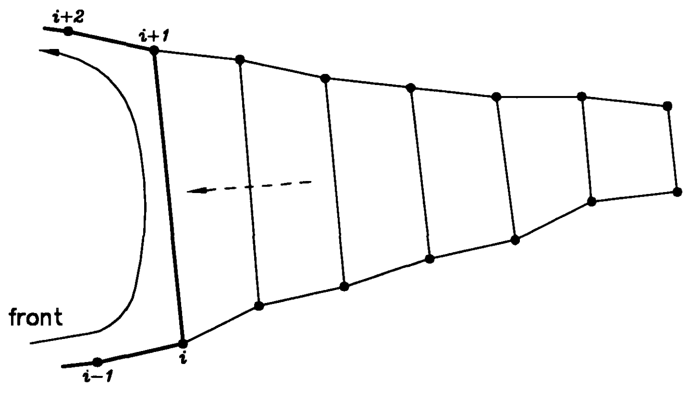

Figure 3-21: Distorted elements that would be generated by the sequential execution of the closing corner routine if the distance between the nodes $i - 1$ and $i + 2$ were not bounded.

After the completion of the closing corner routine, the angles and distances of the nodes $i - 1$ and $i + 2$ are reevaluated.

3.6 Local seaming of nodes

The seaming of nodes is a corrective routine that eliminates small angles in the front as illustrated in Figure 3-22. This function collapses any angle equal to or less than 45 degrees, deletes the node $i + 1$, and evaluates the angles and distances of the nodes in the vicinity of the new position of the node $i - 1$. The function also performs the projection of this node onto the curved surface.

The circular advance of the front deals with corners by creating a new element with the existing nodes. However, for very small nodal angles, the front can intersect itself while approaching the corner, since the check of intersections does not consider the neighbors of a node as mentioned in Section 3.5. Thus, it is necessary to eliminate nodes with small angles by seaming its two adjacent nodes. This procedure creates a new corner with a larger angle as depicted in Figure 3-22.

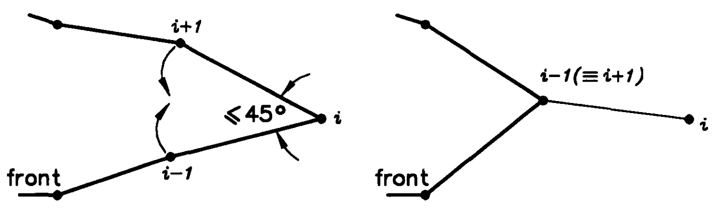

Figure 3-22: Seaming of nodes in the front.

3.7 Smoothing of the front

The front is smoothed if there is any significant variation of the distances between nodes along the front. This variation can be introduced by a modification executed in the front or by a local density refinement. Basically, the algorithm moves nodes in the front attempting to minimize this variation.

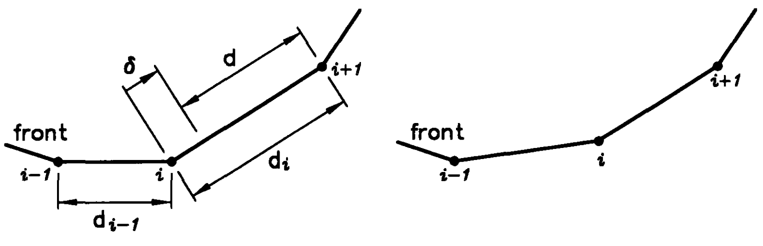

Figure 3-23: Smoothing of the front.

The algorithm performed in this function is executed twice for each node of the front. As depicted in Figure 3-23, it calculates the distance d and compares its value with d_{i-1} and d_i to execute the movement of the node i towards the farther adjacent node ($i-1$ or $i+1$) using the displacement δ:

$$d = \frac{d_{i-1} + d_i}{2} \tag{3.14}$$

$$\delta = \max(d_{i-1},\ d_i) - d \tag{3.15}$$

Note this scheme tends also to smooth the angles in the front. Note also that the projection onto the surface is necessary after the movement of any node.

Once the smoothing of the front has been completed, the angles and distances of the modified front are reevaluated.

3.8 Prediction of intersections and connection of fronts

The intersection of fronts is a necessary feature that enables the generation of meshes over complex geometries. It manages the combination of different fronts until only simple and small fronts exist. As we discuss in Section 3.9, the closure is executed on fronts with fewer than eight nodes, namely, six and four nodes.

The method performs checks of intersections within the front and with other fronts in the same surface of the solid. The check is done before advancing the front, i.e., before the intersections take place. Actually, that is the meaning we want to infer from the term "prediction of intersection". The algorithm foresees the intersections by measuring distances between nodes that, most likely, will intersect if the front advances. It differs from the way *paving* for plane surfaces performs the check of intersections [27],[29],[30]. The latter approach checks the intersections of lines after they have happened which is feasible on the plane, because the computational work necessary to carry it out is relatively small. The same check on a curved surface becomes more expensive.

The prediction of intersection proposed is quite simple. It measures the distance among the nodes of the front and verifies whether there is enough space to advance the front without intersecting. If this check concludes that the distance between two nodes of the front is smaller than the distance necessary to advance the front, the intersection is performed according to one of the procedures described in the next two subsections, otherwise the front is allowed to advance. The same approach is used if the check is performed between two different fronts.

3.8.1 Connection of nodes of the same front

In the event that the check performed by the method predicts intersection between two nodes of the same front, two new fronts are created with the nodes of the original front as depicted in Figure 3-24.

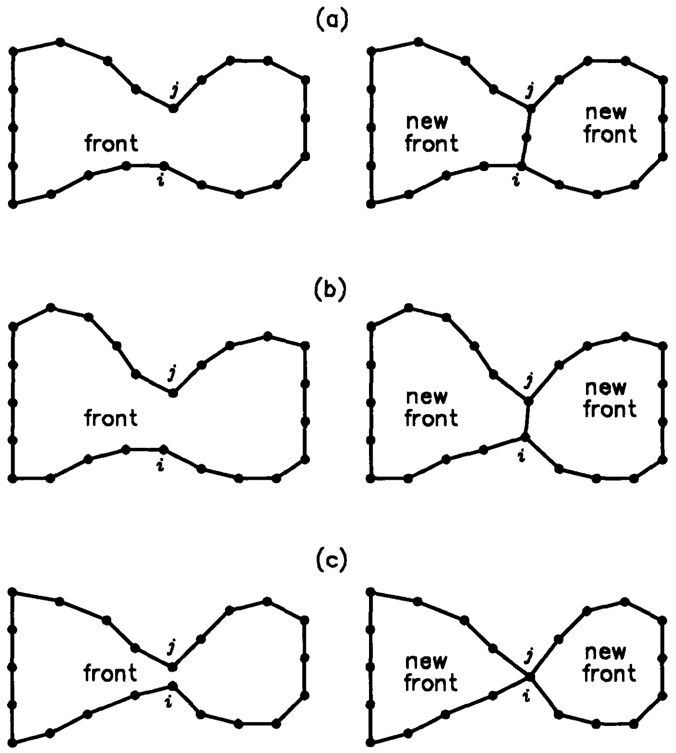

Figure 3-24: Intersection between two nodes of the same front.

The function divides the fronts into two new fronts with even numbers of nodes in order to guarantee the closure using only quadrilateral elements. One of the following routines is executed in order to intersect the front correctly:

- Intersection with creation of a new node – depicted in Figure 3-24*a*.
- Intersection with movement of the two nodes – depicted in Figure 3-24*b*.

- Intersection by seaming the two nodes – depicted in Figure 3-24*c*.

The scheme of Figure 3-24*b* is applied when the number of nodes of the two new fronts is already even, whereas the other two schemes are used when this number is odd. Note the schemes of Figure 3-24*a* and *c* are selected in accordance with the distance between the two points of intersection, nodes i and j. The latter scheme is adopted for shorter distances. However, if the surface has a rectangular-like shape, the scheme of Figure 3-24*a* is adopted in order to avoid the surface being discretized with a nonuniform mesh; i.e., with a node surrounded by a number of elements different from four (see Figure 3-19). The limit in distance that controls the selection between these two schemes is established locally in the mesh, taking into consideration the average element size in the vicinity of the nodes i and j.

3.8.2 Connection of nodes of two different fronts

The second type of intersection takes place when the nodes of two different fronts are predicted to intersect. Typically, this is the case of a surface with one or more holes. To solve this intersection, one of the schemes illustrated in Figure 3-25 is executed to join the two fronts and create a third one with an even number of nodes:

- Intersection with creation of two new elements – depicted in Figure 3-25*a*.
- Intersection with creation of a new element – depicted in Figure 3-25*b*.
- Intersection by seaming the nodes – depicted in Figure 3-25*c*.

In Figure 3-25, the distance between the nodes i and j indicates which scheme shall be selected to join the two fronts. The scheme shown in (c) is selected for small distances, whereas the ones in (b) and (a) are chosen for longer distances. As in the case of intersection within the same front, the limits in distance that govern the selection of these schemes are established locally by considering the average element size in the vicinity of the nodes i and j. This procedure attempts to minimize the element distortion, especially of those elements created to connect the two fronts.

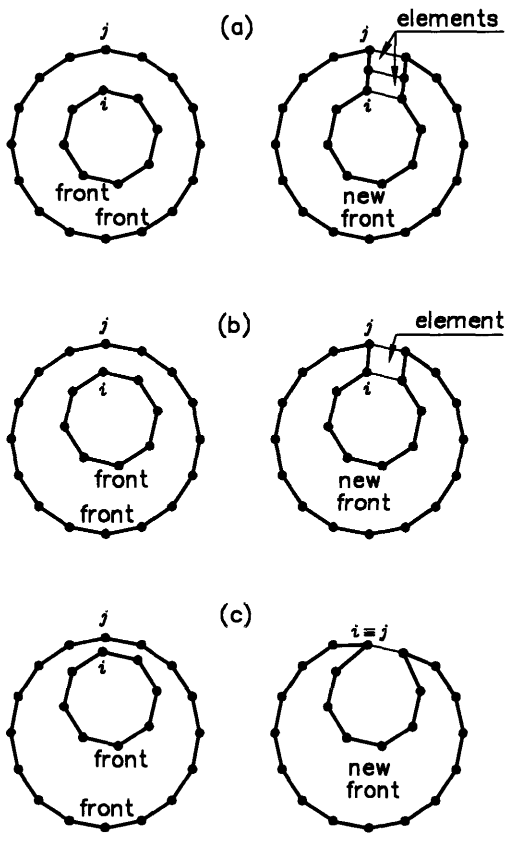

Figure 3-25: Intersection between two different fronts.

Note that, in the intersection of two different fronts, the condition of an even number of nodes in the new front does not drive the selection of the scheme used to intersect the fronts. This condition is always fulfilled for this case, provided that one of the three schemes in Figure 3-25 is used.

3.9 Closure of the front

The check of closure is repeatedly executed in the main body of the mesh generation algorithm. If one of the fronts of a surface has six or fewer nodes, the closure is performed.

The closure concludes the advance of the front either by creating new elements or by seaming the front nodes. The selection of the best approach to close the front is oriented to minimize the element distortion in the mesh.

3.9.1 Closure with the seaming of the front

The seaming of the nodes is a routine that closes the front without creating any new elements. Basically, it can be divided into two cases: front with four nodes and front with six nodes. In the first case, one of the two pairs of opposite nodes must have angles equal to or less than 45 degrees to be considered eligible for the closure by seaming. If this condition is satisfied, the nodes of the other pair of nodes are joined as depicted in Figure 3-26*a*. For the case where the front has six nodes, a similar scheme is used on the three pairs of opposite nodes. If the angles of one of the pairs are less than 90 degrees, and the angles of the other four nodes are greater than 120 degrees, the other two pairs of opposite nodes are seamed as illustrated in Figure 3-26*b*. Note that the nodes seamed are not from the same pair of opposite nodes. In both cases, the points are joined in the middle point between them.

The projection of the nodes onto the surface completes the execution of the closure. Again, note that the projection is executed only in curved surfaces.

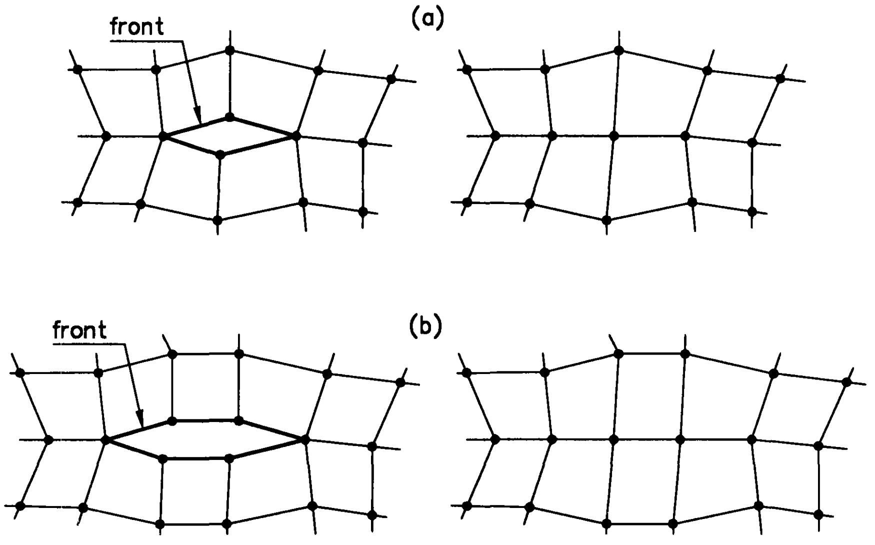

Figure 3-26: Closures by seaming the front: (*a*) front with four nodes; (*b*) front with six nodes.

3.9.2 Closure with generation of new elements

If the condition for seaming the front is not present, the method closes the front with the generation of new elements. In this situation, we also identify the same two cases: front with four nodes and front with six nodes. The solution of the first case is straightforward, being the new element created with the four nodes of the front. For the second case, the method searches the front in order to select the first node of the pair used to divide the front into two new elements. The four possible cases considered in this routine are listed below, starting with the highest priority.

1. Select the node connected to at least one element and with an angle greater than 135 degrees.
2. Select, among the nodes connected to at least two elements, the one with the largest angle.
3. Select, among the nodes connected to at least three elements, the one with the largest angle.
4. Select the node with the largest angle.

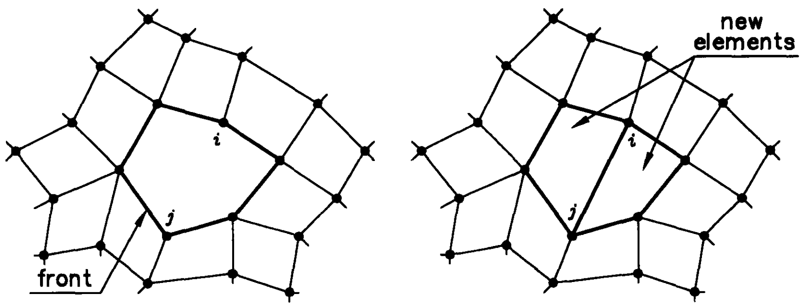

Figure 3-27: Closure with creation of two new elements (front with six nodes).

The goal of this selection is to reduce the distortion of the two new elements to be created and avoid the generation of nodes connected to two and three elements after the closure. The former goal is achieved with the selection of the node with the largest angle in the front (node i of Figure 3-27), whereas the latter one is accomplished with the selection of the node with the minimum number of elements connected to it. In Figure 3-27, the node i satisfies the second of the four possible cases listed above and is selected to close the front along with the opposite node (node j).

3.10 Smoothing of the final mesh

After closing all fronts of the surface, a smoothing routine is applied over the mesh. A standard Laplacian scheme was selected in which the nodal points connected to a given node are considered to calculate the new position of the node (see Figure 3-28).

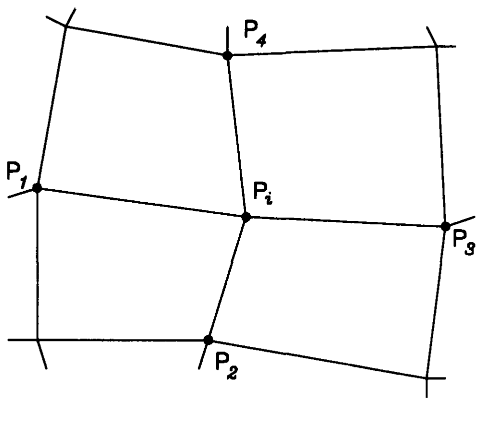

Figure 3-28: Nodes considered for the Laplacian smoothing.

Following Figure 3-28, the new position of the node P_i is calculated with the following stencil:

$$P_i = \frac{1}{n} \sum_{k=1}^{n} P_k \tag{3.16}$$

where $n = 4$ in the case of Figure 3-28.

After the evaluation of the new position of P_i, the node must be projected onto the curved surface. The projection is a computationally expensive routine that accesses AutoCAD via the interpreter, and must be executed for each node in each iteration performed during the smoothing. Therefore, in order to optimize performance, the method takes into account the total number of nodes and the type of the surface in order to set the number of iterations necessary to smooth the mesh. Note that the projection is not necessary if the surface is a plane.

As we indicated in the beginning of this chapter, the smoothing of the final mesh plays a very important role in eliminating the element distortions not corrected with the checks and routines executed to advance the fronts. Excepting the one treated in the cleaning up of the mesh, any remaining distortion is corrected in the smoothing routine.

Despite its simple scheme, the standard Laplacian smoothing has demonstrated its ability to correct the aspect of the mesh satisfactorily. The comparison of the meshes displayed in Chapter 4 with other meshes encountered in the literature, indicates that the additional enhancement gained from more complex smoothing formulations does not pay the price of the computationally expensive schemes associated with them. Note we are comparing unstructured meshes used in finite element analysis. Structured meshes, normally used in finite difference computations, might require elaborated smoothing schemes, such as those described in Subsection 2.1.2.

3.11 Cleaning up of the final mesh

The routine that executes the closure of the front attempts to end the front with elements not very distorted. A node connected to only two elements creates two adjacent quadrilaterals with a triangle-like shape even after smoothing. Unfortunately, this situation can occur far from the closure, while the front is still advancing. For example, successive intersections can, in complex configurations, generate such bad elements. It can also happen with the execution of the seaming-of-nodes routine, as depicted in Figure 3-10*a*.

The cleaning up routine searches the mesh for nodes connected to only two elements as illustrated with the node i in Figure 3-29. Note that nodes in the boundaries are not considered. Once this node is found, the routine deletes it. The routine also deletes one of the two elements connected to this node. Then, the connectivity of the

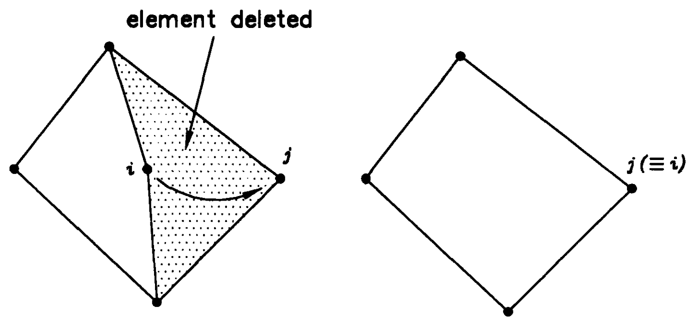

Figure 3-29: Elimination of a triangle-like quadrilateral element of the mesh. Scheme of the cleaning up routine.

remaining element is modified to incorporate the node j. It is represented in Figure 3-29 by the arrow that indicates the movement of the node i to the node j.

Chapter 4

AutoQM: a Program for Automatic Quadrilateral Mesh Generation on the Surfaces of a General AutoCAD Solid

AutoQM is an automatic quadrilateral mesh generation code that implements the algorithm described in Chapter 3 to mesh the surfaces of a general solid created in AutoCAD. The program was written in ANSI-C and uses the AutoCAD Development System (ADS) as an interpreter to access the geometric functions of AutoCAD, especially those related to the Application Programming Interface library (API). These libraries provide essential tools to perform geometric functions, such as the projection of points onto the surfaces of the solid and the evaluation of normal vectors at points of these surfaces.

4.1 AutoCAD graphic interface

We organized the structure of the main functions of the code as part of only one ADS application. It provides the flexibility of transferring the data obtained from the geometry extraction directly to the mesh generation functions, without creating temporary files that slow down the execution of the program.

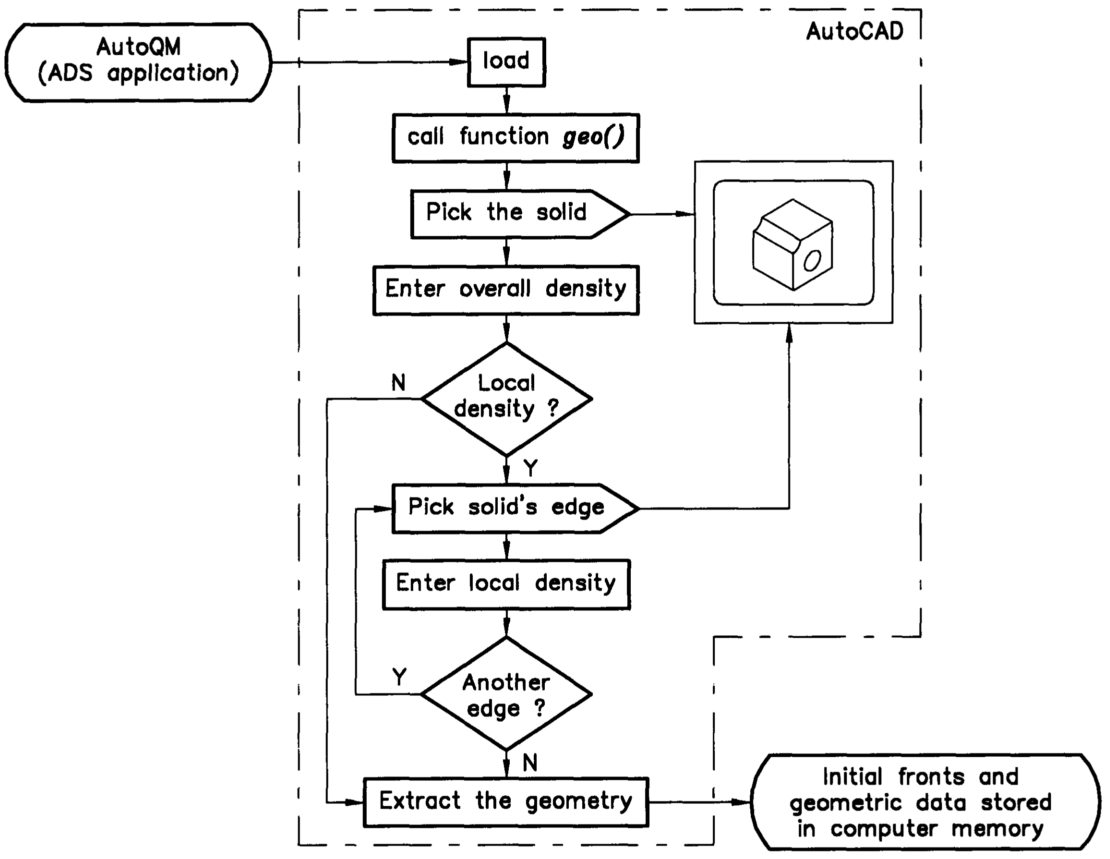

Figure 4-1: Geometry extraction. Scheme of the function *geo()*.

As discussed in Chapter 3, the method is executed in two distinct parts. The first extracts the geometry and creates the initial fronts for each surface of the solid. AutoQM executes this part with the function *geo()* (see Appendix A.1) which asks the user to select the solid to be meshed and to enter the overall mesh density desired, as illustrated in Figure 4-1. This function also provides the feature of imposing different local densities over some of the edges of the solid. It is implemented by asking the user to select each edge with the graphic cursor and enter the local density associated with it. Once the execution of this function is completed, the initial fronts are stored in computer memory and, as already mentioned, no temporary files are created. Part of the information regarding the geometry of the solid is also stored in computer memory in two arrays of structure independent from the AutoCAD database. It is done to speed up the program, since the access of the AutoCAD database is slower than the direct access of an array within the environment of the code.

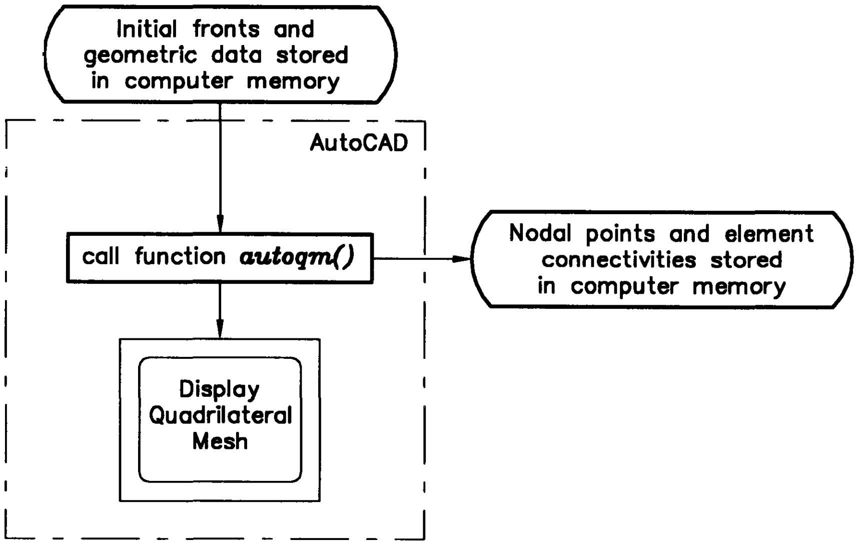
Initial fronts and geometric data stored in computer memory
AutoCAD
call function autoqm()
Nodal points and element connectivities stored in computer memory
Display Quadrilateral Mesh

Figure 4-2: Mesh generation. Scheme of the function *autoqm()*.

The second part is carried out in the function *autoqm()* which automatically generates the mesh over all the surfaces of the solid as shown in Figure 4-2. The user interaction in this part is limited to the choice of displaying the mesh while it is being generated. It provides a fair understanding of how the algorithm is implemented in the code; however, it slows down the whole generation since, every time a modification is introduced in the front, the mesh is partially redrawn. The main body of this function is listed in Appendix A.1. This function calls other C-functions during its execution to construct the mesh as described in Chapter 3. These C-functions are listed in the sections of Appendix A.

4.1.1 Generation of the mesh on curved surfaces

As discussed in Chapter 3, we use the AutoCAD functions to keep track of the surface during the advance of the front. Essentially, the method has to perform two operations to maintain the mesh attached to the surface: the evaluation of the vector normal to the surface and the projection of any new node created or modified in the mesh. The first operation is necessary to establish a local plane in the vicinity of the node in order to evaluate the orientation and the angles of the front, as described in Subsections 3.1.5 and 3.1.6. Executed with the AutoCAD-API function *ap_pt_norm2face(sol_id, sol_face[m].id, pt,TRUE, n)*, the first operation returns the vector n normal to the surface *sol_face[m].id* of the solid *sol_id* at the node *pt*. This function is called in almost every function listed in Appendix A. The second operation maintains the mesh conformed to the surface as the front advances. It is performed with the AutoCAD-API function *ap_pt2face(sol_id, sol_face[m].id, pt1, pt2)* that returns *pt2*, the projection of the node *pt1* onto the surface *sol_face[m].id* of the solid *sol_id* (see function *point_projection(...)* in Appendix A.3).

These operations are key functions in the method. They also provide the possibility of adapting the algorithm to any other CAD package that has these functions incorporated and a development environment similar to ADS.

4.2 Database structure

As the mesh is being generated, the nodes of the fronts are incorporated into the database that contains the element connectivities. The nodes are stored in an array of structure that, basically, contains the sequential numbers of the nodes, their coordinates (x, y, z) and an array with the sequential numbers of the elements to which each node is connected. The quadrilateral elements are also stored in an array of structure containing their sequential numbers in the mesh and the numbers of the four nodes of their vertices. This organization of data allows the algorithm to retrieve the nodal points adjacent to a certain node via the connectivities stored in the two arrays of structure. It is especially useful to execute the smoothing of the final mesh.

The control of nodes in the front is done in another array of structure that stores the node numbers, their angles, the distances between adjacent nodes of the front, and the status labels as defined in Subsection 3.1.4. This array of structure, as well as the ones used to store the nodes and the elements, is listed in Appendix A.2.

These three arrays of structure provide good control of data in the front during the execution of the seaming routine. In this case, the mesh is modified by changing the connectivity of only one of the two quadrilateral elements involved. Similarly, the prediction of intersections takes advantage of these data structures when the intersections are carried out with the seaming of two nodes.

At the end of the mesh generation, the node coordinates and element connectivities can be directly printed out in a file to be used as the input of any finite element analysis.

4.3 Examples

This section illustrates the capabilities of the method and robustness of the program AutoQM with practical examples. The geometries proposed demonstrate the efficiency of the algorithm to advance the front on complex configurations with uniform as well as different mesh densities.

The first example is the intersection of two cylinders with different diameters meshed with uniform density (see Figure 4-3*a*). The cylinder with smaller diameter (30) intersects the other one (dia.=60) with an angle of 60 degrees. The overall density used is 10. Note that the densities and the dimensions are given in units of drawing.

Figure 4-3*b* shows the same configuration of two intersecting cylinders with local density of 2 units of drawing in the edge where the intersection occurs. Note the elevated density gradient between the right-hand edge of the horizontal cylinder and the edge of intersection.

The second example illustrates the capacity of the method to handle different mesh densities in the same front. As depicted in Figure 4-4, the local refinement in the fillet (density 1) migrates to two different densities in the piece, 10 on the right-hand edge and 5 on the opposite edge of the solid.

The third example shows a spherical sector with holes of different shapes and sizes

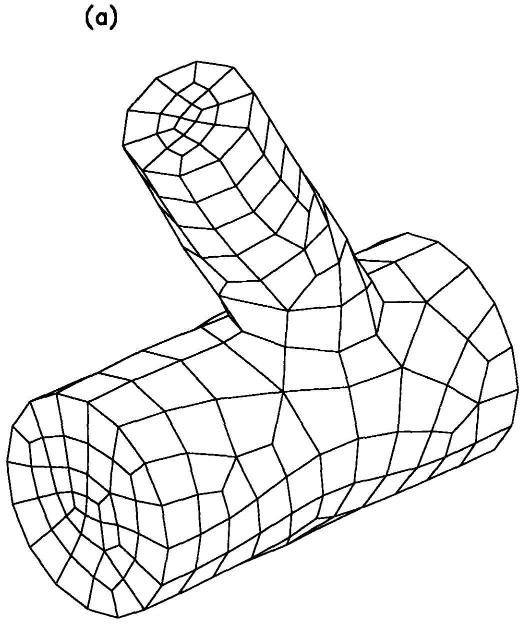

Figure 4-3: Example of a mesh generated over two cylinders intersecting.

(see Figure 4-5). It illustrates the capacity of the method when multiple fronts are present on a surface with curvature in two directions. The overall density in this example is 2 with local density equal to 8 along the border of the spherical sector.

4.4 Discussion of the method

The examples displayed in the previous section demonstrate the capacity of the method to mesh complex geometries with quadrilateral elements. As in *paving* for planar surfaces published by Blacker *et al.*, the quality of the elements of the mesh generated with uniform density is quite good. A minimum of distorted elements are noticed mainly near the closures and intersections [27],[28],[29],[30]. The introduction of local densities increases the number of distorted elements, because frequent corrections of the element sizes in the fronts have to be executed in order to handle the migration among the different densities. Moreover, the occurrence of intersections between two fronts with different densities tends to create even more distorted

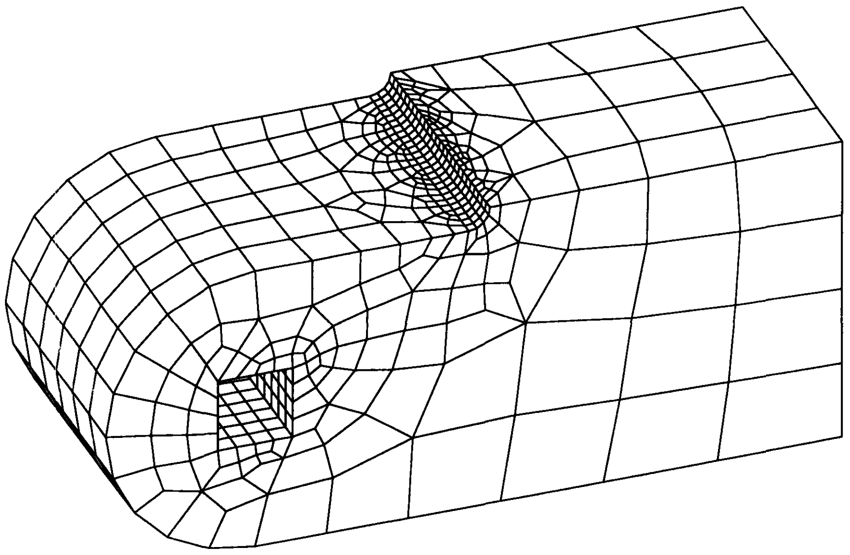

Figure 4-4: Example of local refinement of the mesh in a fillet.

elements.

The extension of the method proposed by Blacker *et al.* to generate meshes on curved surfaces demands the frequent evaluations of the vector normal to the surface and the projection of each new node created or modified in the front. It also requires that the advance of the front be checked *a priori*, i.e., before advancing the front. This latter procedure is important to reduce the time expended in the generation, which is already elevated in *paving* for planar surfaces. In the work of Blacker *et al.*, the average rate of generation is 15 elements per second on a VAX 11/780 machine. This work also mentions that the rate of generation is strongly dependent on the geometry being meshed. Configurations, in which a great number of intersections and seaming of nodes occur, consume more time to generate the mesh than those with simply connected surfaces. In our case, we used two machines to evaluate the rate of generation: a Sun Sparc-5 and a Silicon Graphic Indy workstation. In the example shown in Figure 4-3*a*, 306 elements were generated in 209 seconds on the Sparc-5 and in 129 seconds on the Indy. We have to stress the difficulty to compare CPU time of different machines when not only a C-code is used but also a graphic interface with AutoCAD is accessed. The large amount of CPU time expended in the

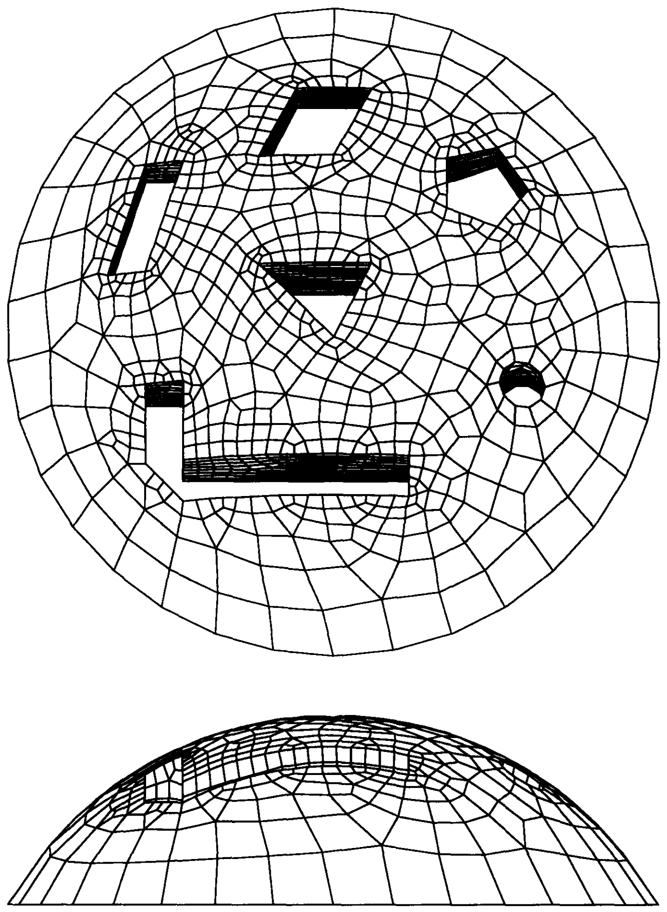

Figure 4-5: Example of a mesh generated with multiple fronts in the same surface of the solid.

execution of the graphic functions of AutoCAD commands the overall performance of the mesh generation. On the other hand, the execution of any graphic function is clearly faster on the Silicon Graphic machine. The same difficulty arises when we compare this result with the one published by Blacker *et al.*, since they also used a different machine. Nevertheless, these results play an important role in the evaluation of results presented in this research.

In the other example presented in Figure 4-3*b*, 1,249 elements were generated in 1,228 seconds on the Sparc-5 and in 744 seconds on the Indy. It displays a rate of generation smaller than the one shown in Figure 4-3*a*. This difference can be explained by the presence of the local refinement, which increases the number of modifications necessary to advance the fronts and, consequently, the whole time of generation as discussed above.

Figure 4-4 displays another example in which AutoQM generates 800 elements in 325 seconds on the Sparc-5 and in 201 seconds on the Indy. Here, we note that the rate increases due to the predominant number of planar surfaces that do not require the projection of points nor the evaluation of the vector normal to surface every time a node is created or modified. It is evident in another simple example tested, in which a cube with a square hole was meshed with 2,200 elements in 17 seconds on a Spark-5 and in 7 seconds on a Indy.

In Figure 4-5, we obtained 2,511 elements in 3,187 seconds on the Sparc-5 and in 1,952 seconds on the Indy. Again, the great number of intersections and other modifications that occur in the front during the generation slows down the execution of the algorithm. Note that part of these modifications takes place on the spherical surface which greatly amplifies the negative effect they have on the performance of AutoQM.

The time consumed in the generation of the mesh seems to be the main limitation of the method. However, this limitation is not due to the algorithm itself but to the graphic interface used, namely the functions used to project points and obtain vectors normal to the surfaces of the solid. As pointed out in the description of the algorithm in Chapter 3, the frequent evaluation of the vector normal to the surface is necessary only on curved surfaces. The same happens with the projection of any new node created or modified during the mesh generation. These functions are executed in AutoCAD and are accessed via the ADS interpreter (graphic interface), which performs poorly if compared with the rest of the C-code of AutoQM. It was verified with the meshes generated on solids bounded with only planar surfaces as mentioned above in the example of a cube with a square hole. In this example the graphic interface is accessed only once for each surface of the solid in order to evaluate its normal vector. In this context, the circular advance of the front has to be mentioned as an important feature introduced to improve the rate of generation. Blacker *et al.*

advance the *paving* front by constructing a sequence of linear rows where the starting and ending points have to be identified. It demands additional analyses before the entire front advances, which is expected to grow as the complexity of the geometry increases. As already discussed in this text, the circular advance of the front requires only the identification of one starting point before advancing the entire front.

Chapter 5

Automatic Hexahedral Mesh Generation: An Advancing Front Approach

In three-dimensional problems, the advancing front method has been successfully used to generate tetrahedral element meshes; however, a lack of schemes using hexahedral elements has been noted. Blacker *et al.* [31] and Stephenson *et al.* [32] have proposed *plastering*, a method that attempts to generate only hexahedral elements by projecting the quadrilateral faces of the three-dimensional front. The generation starts along the boundary where the initial 3D-front is obtained directly from the quadrilateral mesh created to discretize the surfaces of the solid. After advancing the initial 3D-front, the geometric contour is offset inward, which generates the first layer of hexahedrons. This process is repeated until the whole domain has been filled with hexahedral elements.

In this chapter we present an algorithm to automatically generate hexahedral meshes based on *plastering.* This method can only discretize domains in which the exact closure of the mesh is possible. It limits the range of configurations that can be meshed with this method. Actually, we demonstrate that *plastering* is limited to mesh simple geometries. If more complex domains are considered, some intersections and closures can be solved only with non-hexahedral elements.

As already pointed out, the initial front is directly obtained from the 2D-mesh generated on the boundary of the solid using the method presented in Chapter 3. The code developed follows the same structure of AutoQM, being the main function that generates the hexahedral mesh incorporated in the same ADS application that loads AutoQM into AutoCAD. It allows the direct transfer of the quadrilateral mesh generated with AutoQM to the 3D-mesh code by accessing the database stored in memory, without having to write out any temporary files.

5.1 Initial 3D-front

The data necessary to construct the initial 3D-front is extracted from the quadrilateral mesh generated on the surface of the solid. It uses the same database created to store the 2D-mesh in order to control the advance of the front. In addition to the information that defines each quadrilateral element of the 3D-front, the connectivities among the quadrilateral faces are also stored, i.e., for each face f, we store the numbers of its four adjacent faces (see Figure 5-1). Note that quadrilateral elements are called quadrilateral faces in the text, since the word element is widely used for hexahedrons hereafter. This database also stores the angle between adjacent faces following the orientation established for the front. Basically, the orientation of the front indicates that the vectors, normal to the faces, point inward into the domain

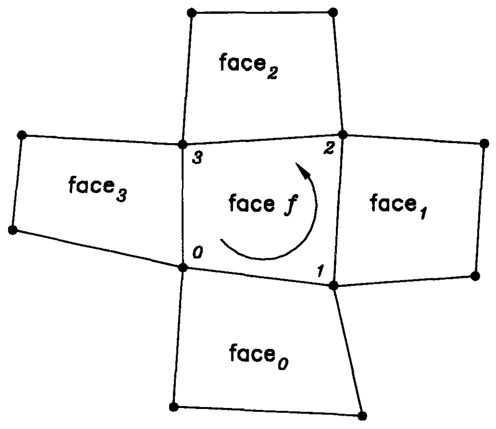

Figure 5-1: Adjacent faces of a given face f of the 3D-front.

bounded by the front (see Subsection 5.1.1).

5.1.1 Orientation of the 3D-front

In the three-dimensional mesh, the orientation of the front represents the direction of the normal vector of each one of its faces. The normal vectors are constructed pointing inward into the domain bounded by the 3D-front. This convention was selected to provide a three-dimensional reference used to advance the front in the right direction.

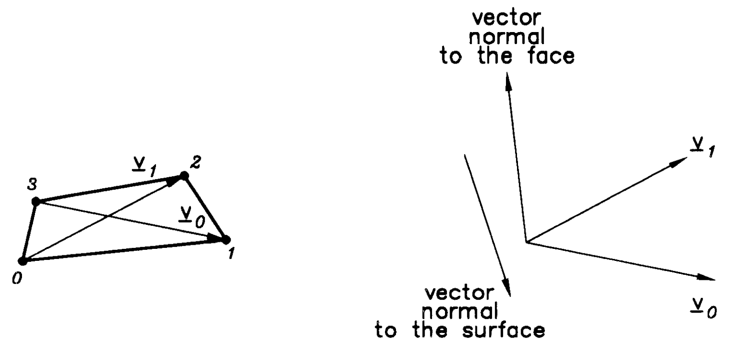

Figure 5-2: Evaluation of the normal vector of a face following the orientation of the 3D-front (face of the initial 3D-front).

As depicted in Figure 5-2, the normal vector is obtained with the product of the vectors defined by the two pairs of opposite nodes of the face, vectors $\underline{v_0}$ and $\underline{v_1}$. In the initial front, the result of this product is compared with the vector normal to the surface, which points outward from the solid. If the inner product of these two vectors is positive (angle less than 90 degrees), the normal vector of the face is multiplied by '-1'. It guarantees that the normal vector always points inward, which is the desired orientation. In the subsequent fronts, the normal vectors are automatically set with the correct orientation by following the direction in which the faces are projected (see Subsection 5.3.1). Note that, for curved faces, the normal vector represents the average plane of the face, which we approximate here by the plane defined with the vectors $\underline{v}_0$ and $\underline{v}_1$. The evaluation of the normal vector using integration over the face is computationally expensive and requires the parametric representation of the face, which is not necessary to generate the mesh.

5.1.2 Angle between two adjacent faces of the 3D-front

The evaluation of the angle between two adjacent faces in the 3D-front employs the same concept of semi-angles used to calculate the angles of the 2D-front (see Subsection 3.1.6). Since the faces are not planar quadrilaterals, we have to select one direction to represent the average plane of the face as illustrated with the vectors $\underline{v}_0$ and $\underline{v}_1$ in Figure 5-3.

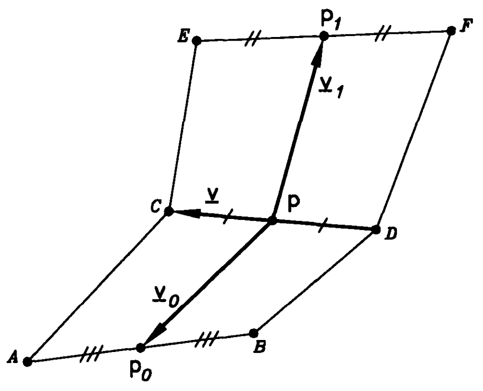

Figure 5-3: Representative directions of adjacent faces for the angle calculation (vectors $\underline{v}_0$ and $\underline{v}_1$).

We cannot simply take the angle between the vector $\underline{v}_0$ and $\underline{v}_1$ as the angle between the two faces, because these vectors are not perpendicular to the vector $\underline{v}$. To make them perpendicular, we execute two vector products that together are equivalent to rotate them in the planes formed by each one of the vectors $\underline{v}_0$ and $\underline{v}_1$, and the vector $\underline{v}$. Taking the vector $\underline{v}_1$ as an example, we calculate $\underline{w} = \underline{v}_1 \times \underline{v}$ which is a vector perpendicular to the plane defined by $\underline{v}$ and $\underline{v}_1$. In the second step, we recalculate $\underline{v}_1 = \underline{v} \times \underline{w}$. After processing both vectors, we normalize them to obtain the configuration depicted in Figure 5-4. The vector $\underline{v}_2 = \underline{v}_1 - \underline{v}_0$ is used to calculate $\underline{t} = \underline{v}_2 \times \underline{v}$, which lies in the plane defined by the vector $\underline{v}_0$ and $\underline{v}_1$. Then, the vector $\underline{t}$ is compared with the resultant of the sum of the two normal vectors of the faces, $\underline{n}$. If the angle between $\underline{n}$ and $\underline{t}$ is greater than 90 degrees, we take $\underline{t} = -\underline{t}$ as the reference for the semi-angle a. Note, this case can happen if the vectors $\underline{v}_0$ and $\underline{v}_1$ are swapped. Finally, we evaluate the semi-angle, a, using the inner product of the vectors $\underline{t}$ and $\underline{v}_0$, and obtain the angle between faces defined as $2a$. As in the 2D-mesh, the use of the vector $\underline{t}$ is necessary since the angles between faces can assume any value in the range 0 to 360 degrees. Angles greater than 180 degrees would be evaluated incorrectly if the inner product of the vectors $\underline{v}_0$ and $\underline{v}_1$ is used directly. If this wrong procedure were used, all angles would lie between 0 and 180 degrees.

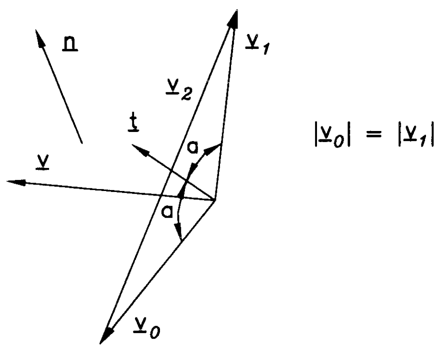

Figure 5-4: Evaluation of the angle between two adjacent faces, $2a$.

5.2 General structure of the algorithm

After setting the orientation and evaluating the angles of the faces, the initial 3D-front is read to advance. The advance of the 3D-front is performed by a process denominated offsetting, which consists of projecting all its faces inward. The projection of a face is a procedure that creates a hexahedral element in such a way that the face of the front seem to be extruded inward. This technique takes into account the adjacent faces and the geometry of the 3D-front in the vicinity of the face being projected. It avoids duplicating faces defined with the same nodes, and prevents the occurrence of notches and overlaps inside the mesh (see Section 5.3).

Since the closure is assumed to be exact, the main body of the algorithm is relatively simple; however, the routine that advances the 3D-front by projecting the faces is quite complex. The general structure of the algorithm is executed in the function *mesh3D()* (see Appendix A.1) following the sequence below.

```
REPEAT until the end of the mesh
     REPEAT for all faces of the 3D-front
          IF the face is enabled for projection THEN
               Project face
               Disable faces connected to two elements
               Evaluate normal vector of the enabled faces
               Set adjacent faces in the vicinity
               Evaluate the angles between faces in the vicinity
     END of loop
END of loop
Smooth the mesh
Check the validity of the mesh
```

A face is considered enabled if it is connected to less than two elements, which means that the face is part of the 3D-front. If the face is connected to two elements, the method disables it, since no projection can be executed in the face.

The main loop increments the label that controls whether the faces of the 3D-front belong to the present front or to the new front being constructed. For instance, the faces of the initial 3D-front receive the label '0' and all the faces created in the first offsetting are labeled '1'. The check of the end of the mesh is positive if all the faces with the new value of the label are disabled, just after the last advance of the front performed.

5.3 Advance of the 3D-front by offsetting

With the initial 3D-front set, the algorithm starts the generation of elements by offsetting the front. This offsetting is accomplished after projecting all the faces of a given front and creating a new layer of hexahedral elements. The name projection indicates that each one of the four nodes of the face generates another node to complete the eight nodes necessary to construct the element. This process takes into account the vicinity of each node to be projected in order to avoid the occurrence of two nodes with the same coordinates, notches, and overlaps inside the 3D-mesh. For example, a face in a vertex of a cube has to project only the node opposite to the vertex to complete the eight nodes required for the new element. In addition, some situations do not demand the projection of any node of the face, because the local configuration already provides the opposite face containing the other four nodes necessary to generate the element.

Figure 5-5 shows that the first face selected to be projected in the front is located in one of the edges of the solid. The next faces to be projected follow the order in which the quadrilateral elements were generated in the 2D-mesh, so that the offsetting is progressively accomplished from the edges to the interior of the solid. This order of generation tends to copy the pattern of the initial front into the next fronts. Note that the faces connected to the edges of the solid, after creating the elements, offset the edges inward and no distortion is introduced in the faces of the next front due to the shrinking of the volume delimited by the 3D-front.

The advance of the front is executed with the function *advance_face_front(...)* listed in Appendix A.23, which is called in the main loop of the function *mesh3D()*.

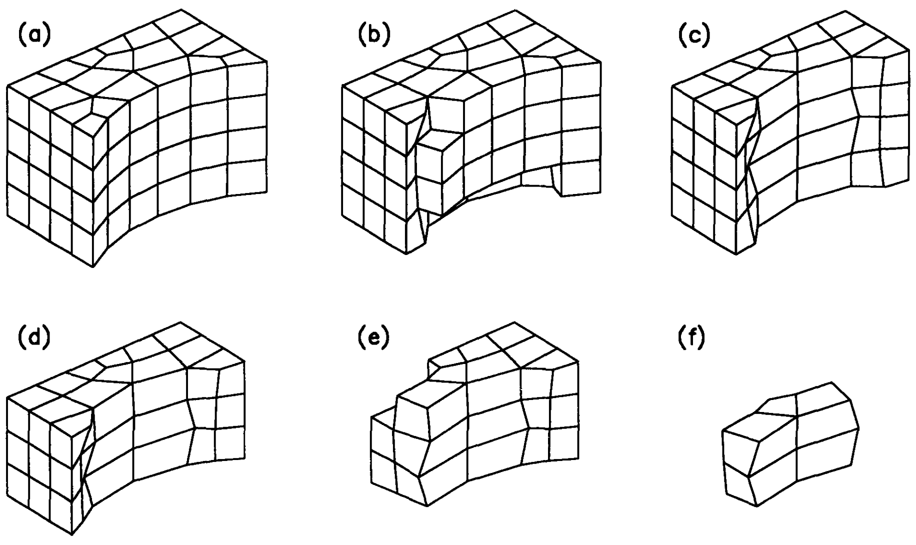

Figure 5-5: Advance of the 3D-front by successive projections of faces (sequence from (a) to (f))

5.3.1 Projection of the faces of the 3D-front

The projection of the faces is the mechanism used in the method to advance the 3D-front. Executed individually for each face of the front, the process starts with the identification of the nodes in the vicinity of the face as depicted in Figure 5-6. Then, the coordinates of the nodes P_i and V_i are compared in order to figure out the shape of the front around the face f and decide how to project the face.

The method searches for three configurations before creating any new node by projection. The first configuration deals with a vertex-like geometry in which one of three faces is to be projected (face f in Figure 5-7). A fourth face, opposite to the face f, is also present as depicted in Figure 5-7.

The identification of the first configuration starts with the check for coincident nodes in the vicinity of the face f. In Figure 5-7, the check is considered positive if the expression $< P_5 \equiv P_6$ and $P_7 \equiv V_5 >$ is true. This check indicates that the eight nodes necessary to form the new element are already available. In this case, two

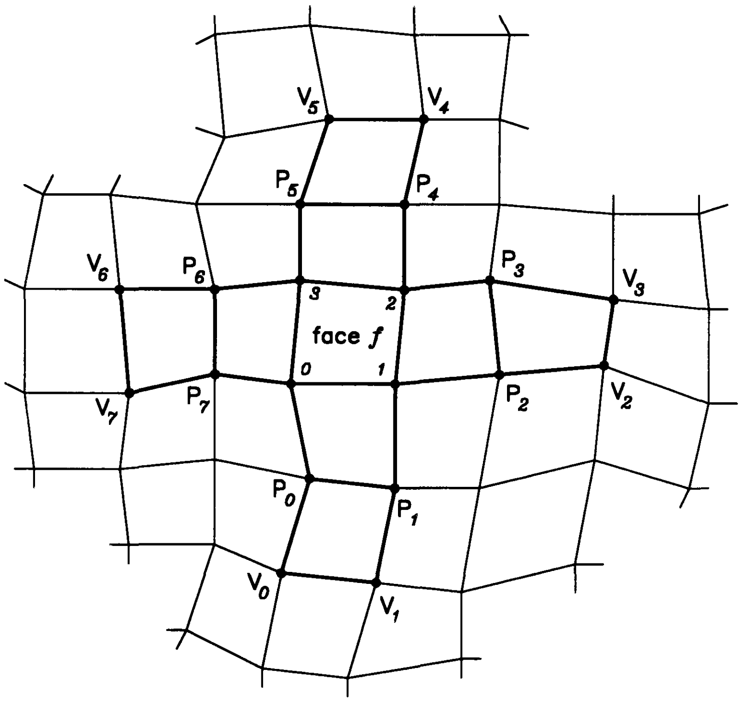

Figure 5-6: Nodes in the vicinity of the face to be projected (face f).

new faces are created to accomplish the projection of the face f. Note, however, that these other two faces may already exist, which reduces the mechanism of projection to a simple updating of connectivities in these faces.

The faces needed to complete the new element may already exist but may not be totally connected to the nodes of the new element as illustrated in Figure 5-8. In this situation, the face is seamed with the new element; i.e., node P_2 is deleted and its connectivity transferred to node V_4.

The second configuration is similar to the first one in which the face opposite to the face f does not exist, as depicted in Figure 5-9. This configuration is searched only if the first configuration is not present.

The check executed to detect the second configuration involves the coincidence of the nodes P_5 and P_6 displayed in Figure 5-9. It also requires that the angles between the face f and the two adjacent faces (faces 2 and 3) be equal to or less than 135

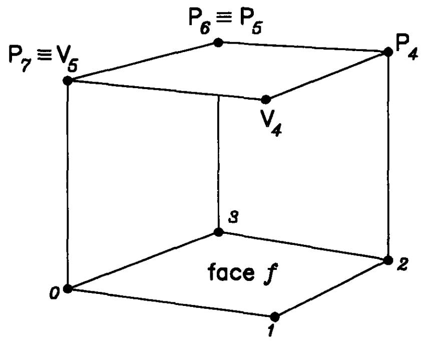

Figure 5-7: First configuration searched for the projection of the face f.

degrees. If the faces 0 or 1 are not connected to the faces 2 and 3, the node 1 is projected to create the eighth node of the element. The node is projected toward the direction of the resultant of the sum of the normal vectors of the faces connected to the node (see vector $\underline{s}$ in Figure 5-10). The vector actually used for projection, $\underline{v}$, is obtained by normalizing the vector $\underline{s}$ with the average distance between the node P_i

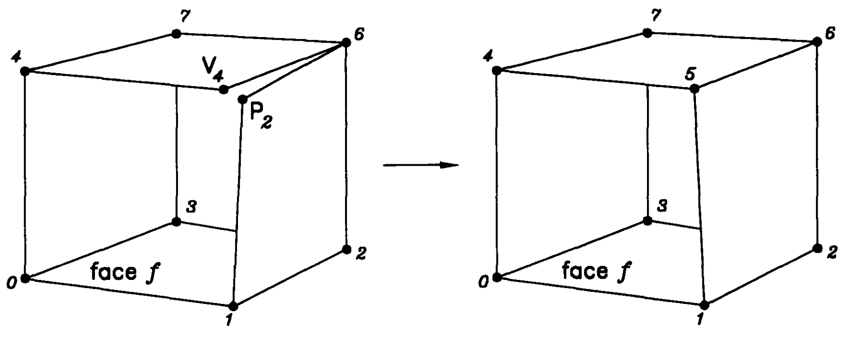

Figure 5-8: Seaming of a face with a new element created by projection.

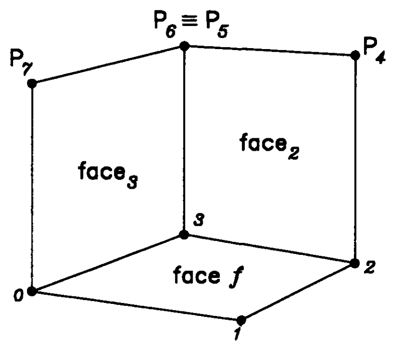

Figure 5-9: Second configuration searched for the projection of the face f.

and the nodes connected to it (P_1 to P_5):

$$\underline{v} = \frac{\underline{s}}{|\underline{s}|} \frac{1}{n} \sum_{k=1}^{n} \text{dist}(P_k,\ P_i) \tag{5.1}$$

where $n = 5$ for the case depicted in Figure 5-10.

The algorithm also measures the angle between the vector $\underline{v}$ and the vectors formed with the segments delimited by the node P_i and the adjacent nodes, P_1 to P_5. If one of these angles is equal to or less than 135 degrees, the adjacent node associated with this angle is taken as the projected node P_j.

The third configuration is searched if none of the two previous configurations is present. It consists of three adjacent faces with angles equal to or less than 135 degrees as depicted in Figure 5-11. This configuration is solved with the creation of three new faces, if they do not yet exist.

If none of the three configurations is present in the vicinity of the face f, the two nodes not connected to the adjacent face with angles equal to or less than 135 degrees (face 2) are projected as illustrated in Figure 5-12. Note that the face f has always, at least, one adjacent face that satisfies this condition, because the first face to be projected matches it, and all the subsequent projections are executed in faces adjacent to one or more elements just created.

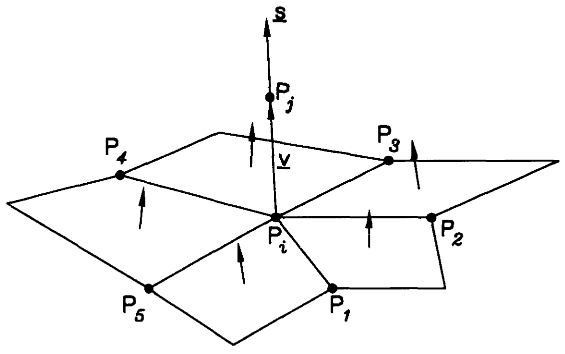

Figure 5-10: Projection of a node.

The configurations proposed in this text to project the faces cover all the possible cases in the front, except those in which intersections and inexact closures occur. However, as we discuss later in this chapter, the method proposed in [31],[32] cannot solve all inexact closures, or even some intersection cases, with all hexahedral elements.

The code developed to carry out the projection is listed in the function *projec-*

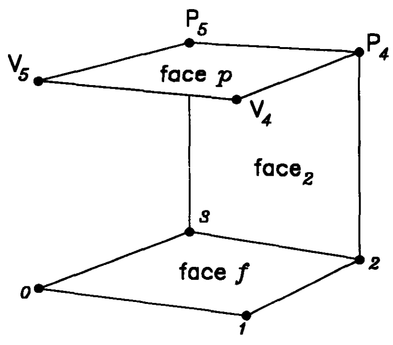

Figure 5-11: Third configuration searched for the projection of the face f.

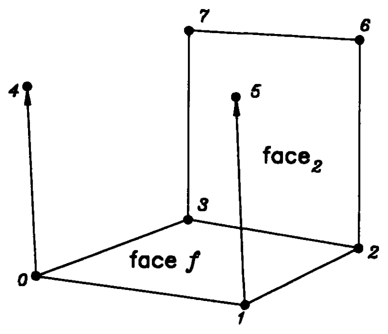

Figure 5-12: Projection of the face not covered in any of the three basic configurations initially searched for the projection of the face f.

tion(...) (see Appendix A.22). It is an auxiliary routine executed in the function *create_element(...)* (see also Appendix A.22), which is called inside the function *advance_face_front(...)*.

5.3.2 Smoothing of the hexahedral element

The shape of the element created with the projection of a face must follow the geometry of the 3D-front in the vicinity of the face. In a vertex of a cube-like geometry, for instance, the method shall create three other faces, in such a way that the element generated is, as much as possible, similar to a parallelogram. To comply with this requirement, the method applies a correction to each node projected, attempting to reproduce the shape of the existing faces in the new faces created to form the element.

The correction of the projection of the node is a routine that deals with two separate cases. The first is applied when only one of the faces adjacent to face f is used during the construction of the new element, as depicted in Figure 5-13. This case is solved with the rotation and normalization of the vectors created for the projection (vectors $\underline{04}$ and $\underline{15}$). The rotation is performed within the planes 014 and 015 in order to copy the shape of the face 2 and preserve, partially, the direction of the

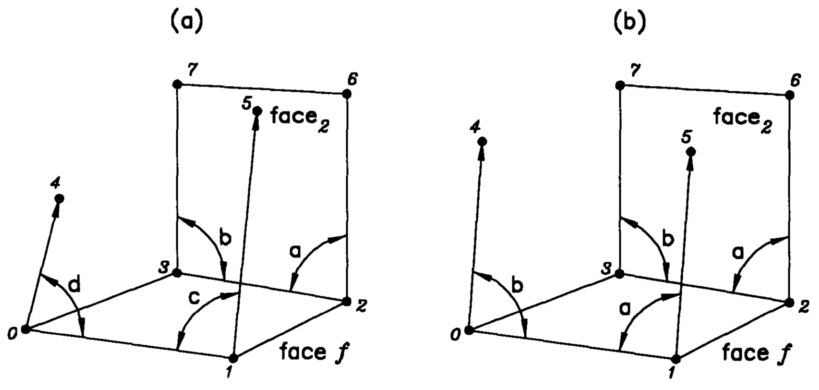

Figure 5-13: Smoothing of an element created with the projection of two nodes.

projection. The magnitudes of these vectors are normalized with the lengths of the vertical edges of the face 2 (segments $\underline{37}$ and $\underline{26}$). This routine is executed with the function *smooth_one_face(...)* listed as an auxiliary function in Appendix A.22.

The second case is used for the node created to solve the projection of the face *f* illustrated in Figure 5-14. Note this is the second configuration described in Subsection 5.3.1. In this case, the projection is corrected by rotating the vector $\underline{15}$, so that the angles *c* and *d* are set equal to *a* and *b* respectively, as depicted in Figure 5-14*b*. The correction is completed after normalizing this vector with the average length of the segments $\underline{04}$ and $\underline{26}$. This routine is executed in the function *smooth_two_faces(...)*, which is also listed as an auxiliary function in Appendix A.22. Both functions are called inside the function *create_element(...)*.

5.4 Exact closure of the 3D-front

The exact closure is the conclusion of the mesh generation with all hexahedral elements. It happens, for instance, in a cube-like geometry with equal mesh patterns in the opposite surfaces of the solid as in the example illustrated in Figure 5-5.

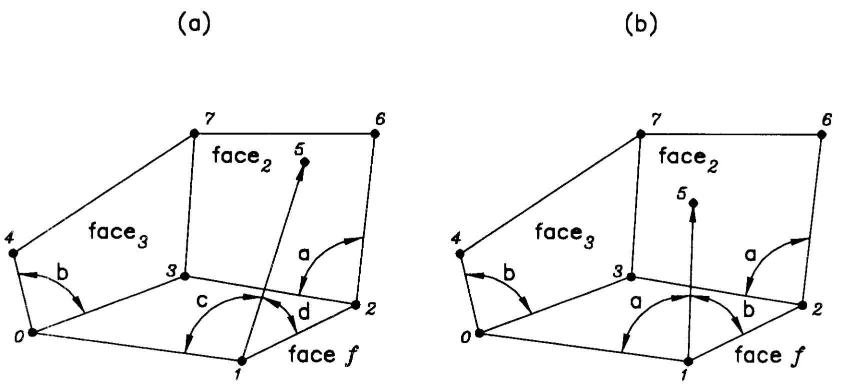

Figure 5-14: Smoothing of an element created with the projection of one node.

To better understand the mechanism that leads to the exact closure, we have to visualize that the contribution of the adjacent faces in the generation of an element by projection increases until no new face is necessary to construct the element, which happens in the last element of the exact closure. On the other hand, we have to notice that the intersection of the 3D-front demands simple patterns in order to provide the correct configuration for the generation of only hexahedrons. If none of these conditions exists, the 3D-mesh can be completed only with an inexact closure.

The inexact closure requires the use of wedge-like polyhedrons, namely pyramids and pentahedrons. This situation happens, for instance, when two sectors of the 3D-front with different patterns and/or densities are present in the closure. Figure 5-15 illustrates three examples of inexact closure with the types of polyhedrons we have mentioned: a wedge (a), a pyramid (b), and a pentahedron (c). In more complex closures, the use of tetrahedrons might be necessary since they are the only polyhedron that guarantees the closure in any configuration.

If only wedges as the one shown in Figure 5-15a occur in the inexact closure, the mesh can be transformed to an all hexahedral mesh by collapsing the nodes 0 and 2 of the wedge and, consequently, the row of elements connected to the face 0123, in a

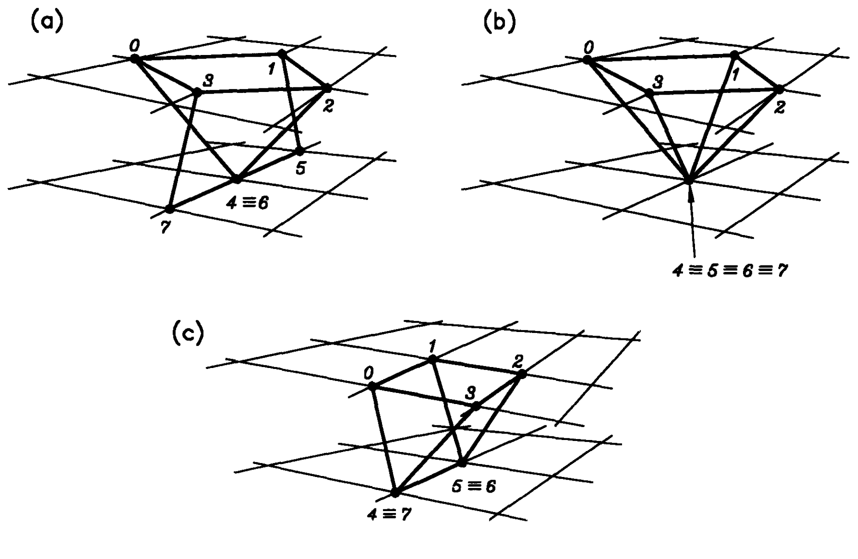

Figure 5-15: Examples of non-hexahedral elements present in inexact closures.

sequence that will end up with an element in the surface of the solid. However, the pyramid in Figure 5-15*b* cannot be solved, even with the additional collapsing of the nodes 1 and 3, which generates triangles in the upper sector of the front. Similarly, the pentahedron depicted in Figure 5-15*c* cannot be solved. If we attempt to collapse the nodes 0 and 3 along with the nodes 1 and 2, the two adjacent rows of elements also have to be collapsed and, unless there is some symmetry, the entire mesh can be destroyed.

The same case can occur in the intersection of the 3D-front. Let us take as an example, the generation of the mesh in a solid that after advancing the 3D-front a certain number of times meets a local configuration of five faces connected in a circular shape as depicted in Figure 5-16*a*. The method generates an element using three of the five faces and a pentahedron remains as the only possible solution to complete the intersection.

In the light of this discussion, we are led to conclude that the inexact closures and the intersections of the 3D-front are the main limitations of the method proposed in [31],[32]. The absence of geometric properties that guarantee the exact closure

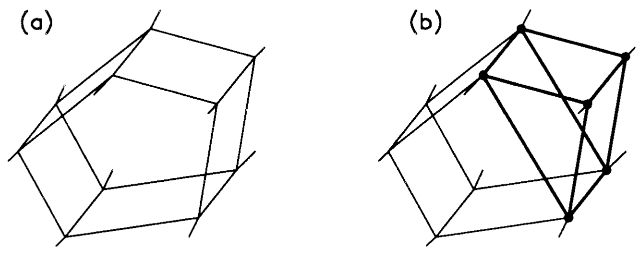

Figure 5-16: Intersection with generation of a pentahedron.

makes any algorithm that proposes to manage the advance of the 3D-front in order to create the configuration necessary to complete the mesh with all hexahedral elements extremely unreliable.

5.5 Smoothing of the 3D-mesh

The smoothing of the hexahedral mesh is similar to the scheme presented in Section 3.10 for the two-dimensional mesh, in which a standard Laplacian scheme was used. It consists of moving each node of the mesh, P_i, to the geometric center of its adjacent nodes, P_k:

$$P_i = \frac{1}{n}\sum_{k=1}^{n} P_k \tag{5.2}$$

where n is the number of adjacent nodes.

The smoothing routine is essential to the check of valid mesh presented in Section 5.6. It must be executed before this check in order to guarantee that all the elements are convex hexahedrons.

5.6 The check of valid mesh

The check of valid mesh consists of comparing the exact volume of the quadrilateral mesh, generated along the boundary of the solid, with the sum of the exact volume of each hexahedral element. A valid mesh displays the same value for these two volumes. To calculate the exact volumes of the solid bounded by the 2D-mesh, we use the volume created with the projection of each quadrilateral onto the coordinate plane xy as depicted in Figure 5-17*a*.

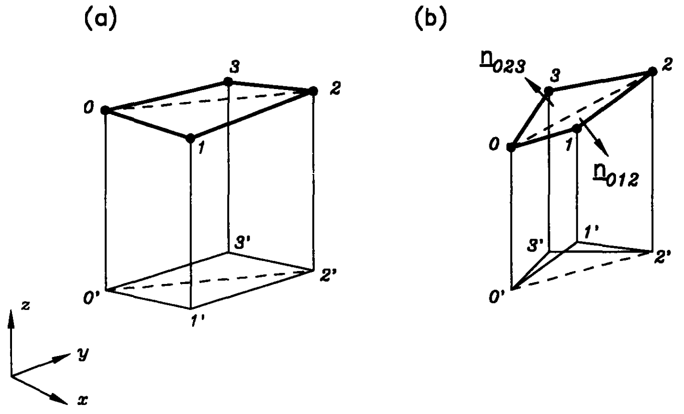

Figure 5-17: Volume formed by the projection of a quadrilateral face onto the coordinate plane xy.

The volume of the projection of a quadrilateral face is calculated as the sum of the two pentahedrons formed by the division of this volume with the dashed lines displayed in Figure 5-17. This procedure is necessary to handle the case where the projection is a folded quadrilateral as illustrated in Figure 5-17*b*. In this case, the volume created with the projection of the triangle 012 shall be added to the total volume whereas the volume related to the triangle 023 shall be subtracted from it. Note we represent the points 1 and 3 with the coordinates $z_1 < z_3$. We use the value of the z coordinate of the normal vector of each triangle to determine whether the volume of its projection must be added or subtracted from the total volume. If the z coordinate of the normal vector is negative (vector $\underline{n}_{012}$) the volume is added, otherwise it is subtracted (vector $\underline{n}_{023}$). If $z = 0$, the volume is not considered for the sum. The normal vector of each triangular plane (vectors $\underline{n}_{012}$ and $\underline{n}_{023}$) points inward into the solid, following the same convention used for the normal vectors of the faces of the 3D-front described in Subsection 5.1.1. Note this approach is similar to a numerical integration over the volume, using the quadrilateral mesh as a partition.

The volume of each pentahedron is calculated as the sum of the volumes of the three tetrahedrons into which it can be subdivided. In Figure 5-17*a*, the pentahedron formed with the projection of the triangle 012 can be subdivided into the tetrahedrons 0121', 20'1'2', and 020'1'. To obtain the volume of a tetrahedron, we use the vectors created with its four vertices as depicted in Figure 5-18.

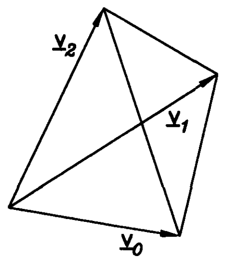

Figure 5-18: Vectors used to calculate the exact volume of a tetrahedron.

The volume of the tetrahedron is then obtained with the inner product of the vectors $(\underline{v_0} \times \underline{v_1})$ and $\underline{v_2}$, where $(\underline{v_0} \times \underline{v_1})$ indicates the vector product of $\underline{v_0}$ and $\underline{v_1}$:

$$V_{tetrahedron} = \frac{1}{6} \left((\underline{v_0} \times \underline{v_1}) \cdot \underline{v_2} \right) \tag{5.3}$$

The volume of the solid represented as the sum of its hexahedral elements is calculated with the subdivision of each hexahedron into five tetrahedrons as illustrated in Figure 5-19. The volume of each tetrahedron is evaluated with equation 5.3.

In order to allow the comparison between the volume of the quadrilateral mesh and the volume resulting from the sum of the hexahedral elements, the nodes used to fold each quadrilateral face of the 2D-mesh must be the same used along the contour of the hexahedral mesh. Similarly, each face inside the hexahedral mesh must have only one folding line. The folding nodes are those connected with a dashed line as illustrated in Figures 5-17 and 5-19. They are established by a simple check of connectivity

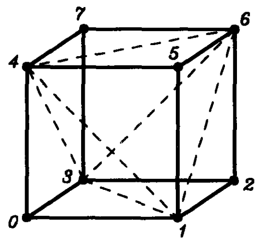

Figure 5-19: Subdivision of a hexahedron into five tetrahedrons: 0134, 1236, 1456, 3467, 1346.

among the nodes of the 3D-mesh. Starting with a randomly selected node, we set its type to folding node and all its neighbors to non-folding type. Note that a folding node is always connected to non-folding nodes and vice-versa (see Figure 5-19). Note also that this procedure substitutes a parametric representation of the faces which requires a more complex implementation.

As mentioned in Section 5.5, the smoothing of the 3D-mesh must be executed before the check of valid mesh in order to provide only convex hexahedrons for the volume calculation. If the hexahedron is not convex, the subdivision required to represent its volume differs from the one adopted.

The 3D-mesh shown in Figure 5-5 was checked and the following values were obtained:

$$V_{2D-mesh} = 8732.019104643110040 \tag{5.4}$$

$$V_{3D-mesh} = 8732.019104643104583 \tag{5.5}$$

The difference of 5.457×10^{-12} demonstrates the efficiency of the scheme. Some larger numerical differences are expected with the increasing of the number of elements in the mesh.

The check of valid mesh is complemented with the verification of the connectivity of each face of the 3D-mesh. A face has to be connected to two and only two elements inside the 3D-mesh, which means that the faces on the surface of the solid are not considered. It guarantees that there are no notches and overlaps inside the 3D-mesh. This complementary procedure provides the information necessary to analyze the values $V_{2D-mesh}$ and $V_{3D-mesh}$. If no error is detected in the connectivities of the faces, the difference is due to numerical imprecision, and the mesh is considered valid. This conclusion is drawn by noticing that, if the connectivities of the faces are correct, the check of valid mesh detects only notches and overlaps that add or subtract volumes with the magnitude of a hexahedral element.

The auxiliary functions used to check the 3D-mesh are listed in Appendix A.25. The volume bounded by the quadrilateral mesh and the volume resulting from the sum of the hexahedral elements are calculated with the functions *volume_of_2D_mesh(...)* and *volume_of_3D_mesh(...)*, respectively. The verification of connectivities among the faces is carried out in the function *check_two_elements_per_face()*.

5.7 Closure uncertainty: comparison with quadrilateral meshes

The robustness of any quadrilateral mesh generation method can be achieved only with the existence of a geometric condition that guarantees, unconditionally, the mesh execution. In transfinite interpolation, the domain discretization relies on the mapping transformation. The advancing front method uses the condition of an even number of nodes in the front to guarantee the closure with all-quadrilateral elements. The transformation from triangles is based on the fact that any triangle can be subdivided or combined to form quadrilateral elements. Geometric decomposition relies on the condition of the secondary method used to mesh each one of its subregions as discussed in Chapter 2.

Once the condition that guarantees the execution of the mesh on any geometry has been identified, the development of the method is concentrated in programming issues, such as database construction and smoothing techniques to generate a mesh with good elements. Note, however, that if the method attempts to create the condition to complete the mesh while it is being generated, its robustness is compromised. When the third dimension is introduced, the existence of such a geometric condition is even more necessary.

The advancing front method has been used to generate tetrahedral meshes in general solids. The geometric property that guarantees the division of any polyhedron into tetrahedrons is the cornerstone of the technique. In this method, no matter where the front starts – on the boundary or in an interior point – the closure is always possible. However, if hexahedral meshes are considered, the number of examples in which the closure with all hexahedral elements is not possible, and the unstructured nature of the generation indicate that the condition to guarantee the closure is limited to very simple geometries as the one illustrated in Figure 5-5.

In the papers that proposed *plastering* as an automatic hexahedral mesh generation, no condition that guarantees the closure was indicated [31],[32]. Actually, the solution of the mesh termination was postponed to further developments. With the study performed in the present text, we conclude that the method cannot solve all closures and intersections in the front with only hexahedral elements, being limited to mesh simple geometries where the transfinite interpolation schemes are more efficient.

5.8 Discussion of methods for hexahedral mesh generation

Mapping techniques have been the mainstream of the 3D-mesh generation methods using hexahedral elements due to their relatively simple computational implementation and good performance. However, to mesh complex geometries, additional techniques are necessary to subdivide the domain into simple polyhedrons [15],[16]. These techniques perform poorly in fully automated schemes, especially when a complex configuration is considered. A similar approach used to tackle complex geometries is the topological representation of the solid in the natural domain, in which the correlation between the boundary of the topological geometry and the geometric contour of the physical domain has to be carried out manually [11],[12]. Another important issue in these approaches is the local refinement of the mesh. Hoyte proposed a scheme called *cut & glue* to generate transitional meshes and allow different densities in adjacent subregions [46]. Although many improvements have been incorporated into the method, the fully automated discretization of general solids with hexahedral elements is still a goal to be reached.

As discussed in the previous section, the robustness of a mesh generation method must rely on a condition that guarantees the construction of the mesh in any general configuration. In the light of this background, we investigate two methods to obtain hexahedral meshes. The first method is based on the transformation from tetrahedral meshes, which are generated with well-published methods that use the fact that any polyhedron can be subdivided into tetrahedrons to guarantee the generation the mesh.

Grid superposition and advancing front methods have dominated the tetrahedral mesh generation. Lo in [47],[48] used the advancing front method in an approach that starts by meshing the boundary with triangles attempting to generate tetrahedrons as equilateral as possible. Johnston also discretizes the surface of the solid before advancing the front in an approach that offsets the 2D-mesh inward [49]. The standard grid superposition, also known as *octree*, has been published by Kela and his colleagues [50],[51],[52]. Similar approaches were proposed by Shephard and Georges in [53] and Buratynski in [54].

The use of the transformation from tetrahedrons requires that the algorithm adopted to generate the initial mesh be able to use coarse densities, because the transformation reduces the density to half of the initial value set. It shall not be seen as a limitation since finer meshes are often desired for improving the accuracy of the solution. Besides, the tetrahedral mesh generation method can achieve fairly coarse densities.

The advancing front method using tetrahedral elements tends to generate uniform meshes with good elements (i.e., nearly equilateral tetrahedrons) throughout the domain. It is an important feature of the method as far as tetrahedral meshes are

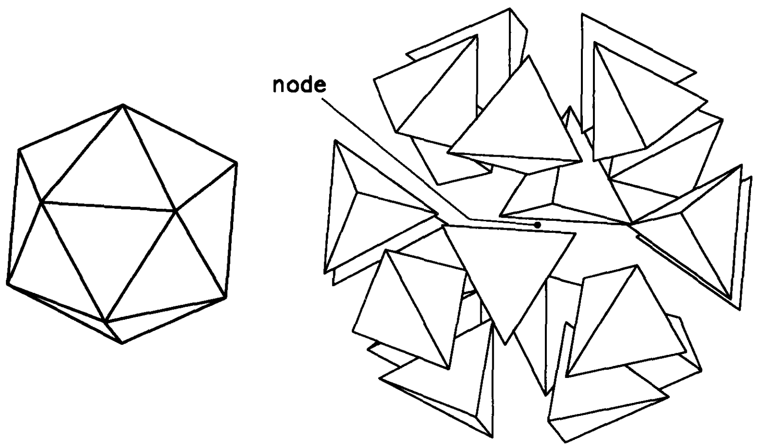

Figure 5-20: Typical node of a tetrahedral mesh generated with the advancing front method. Node with 20 adjacent tetrahedral elements.

concerned; however, it is undesirable in transformation schemes dedicated to generate hexahedral elements. In the advancing front method, a typical node is surrounded by an average of 20 *quasi* equilateral tetrahedrons as depicted in Figure 5-20. This node will also have 20 adjacent hexahedral elements after the transformation, which is 2.5 times as much as the number of elements connected to a node in an ideal hexahedral mesh, i.e., eight adjacent elements per node. On the other hand, the new nodes created inside each tetrahedron are surrounded by 4 hexahedrons which is not a good pattern either.

The second method investigated uses grid superposition to overlay a uniform patch of hexahedrons over the geometry and uses hexahedrons and some pentahedrons, pyramids, and tetrahedrons to connect the mesh to the boundary. If we consider only the initial hexahedral elements generated, we have a core mesh that covers a great part of the volume of the solid as depicted in Figure 5-21.

As already mentioned, the problem of connecting the contour of the core mesh and the boundary of the solid can be solved with hexahedrons and some pentahedrons, pyramids, and tetrahedrons. This method can also be structured to maximize the number of hexahedrons generated. As a result, the use of grid superposition, combined with the generation of non-hexahedral elements in some points along the

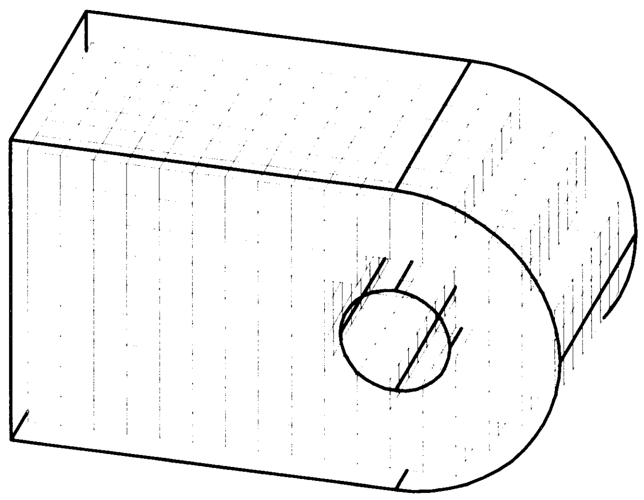

Figure 5-21: Core mesh of hexahedral elements generated with grid superposition.

boundary is, in certain aspects, better than *plastering* with closures and intersections solved also with non-hexahedral elements. The argument commonly used that *plastering* creates good elements along the boundary cannot be taken absolutely. The existence of bad elements in the closure and in other points of the quadrilateral mesh ends up with bad hexahedral elements on the surfaces. Here we raise some points that deserve investigation, namely the comparison between bad hexahedral elements and mixed meshes with some non-hexahedral elements along the boundary, similarly to what has been done to compare the performance of isoparametric elements [2],[3],[4],[5],[6]. Another aspect that must be taken into consideration is that *plastering* generates regions with a great number of non-hexahedral elements randomly distributed throughout the mesh, whereas grid superposition provides some control of their distribution in the mesh. Grid superposition is a boundary sensitive method, which means that the rotation of the domain in respect to the uniform grid changes the final mesh obtained. This feature can be used to create very good elements in sectors of interest along the boundary by setting the uniform grid parallel to the planes of these sectors.

Another approach consists of associating grid superposition with the introduction of the first layer of hexahedral elements generated with *plastering*. It offsets the boundary inward, creating only hexahedral elements on the surfaces of the solid (see Figure 5-22). The connection of the *plastering* mesh with the contour of the core mesh still demands the use of some non-hexahedral elements; however, it takes place one layer underneath the boundary. Note this approach requires that the mesh density distribution in the 2D-mesh allows the construction of the first layer of hexahedrons without intersections.

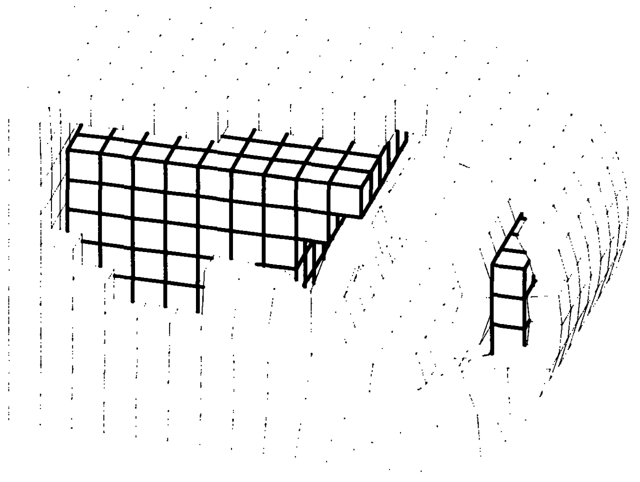

Figure 5-22: Grid superposition associated with the first layer of hexahedral elements generated with *plastering*.

It is important to note that the use of grid superposition, associated with transformation from tetrahedrons to achieve a fully hexahedral mesh, generates elements worse than those obtained if the initial mesh were constructed with the advancing front method, discussed previously in this section. Due to the continuity requirement between the most external elements of the core mesh, and to the non-hexahedrons necessary to connect the core mesh to the boundary, the hexahedrons of the core mesh cannot be simply subdivided into eight regular hexahedral elements. Actually, each one of these hexahedrons must be subdivided into five tetrahedrons which, after being transformed, end up with 20 hexahedrons. As a result, approximately half of the nodes that define the initial core mesh will be connected to 32 hexahedrons, which is worse than the 20 hexahedrons obtained with the advancing front method. Note that a different result is obtained if grid superposition is associated with transformation from triangles to generate quadrilateral meshes. In two-dimensional problems, the connectivities among the elements permit that each quadrilateral of the core mesh be subdivided into 4 regular quadrilateral elements.

Chapter 6

Comments and Conclusions

In this thesis we considered the generation of 2D and 3D-meshes. Regarding the 2D-mesh generation, the method presented is able to automatically discretize general surfaces using only quadrilateral elements. The meshes generated in the examples discussed illustrate the capability of the algorithm to handle different mesh densities in simply connected surfaces as well as in configurations with multiple non-connected boundaries. The examples also revealed the efficiency of the algorithm to generate meshes in the presence of high density gradients. The good quality of the elements is another characteristic of the mesh generated. The few unavoidable distorted elements generated occur in regions far from the boundaries of the surfaces. These distortions, however, are minimized in regions of low density gradients of the mesh.

The performance of AutoQM varies with the complexity of the geometry being meshed. The rate of generation is drastically reduced when curved surfaces are meshed, since the evaluation of the vector normal to the surfaces and the projection of any new node created or modified in the mesh are frequently executed. The improvement of the code shall necessarily demand more efficiency for these geometric operations and, consequently, better performance of the graphic interface, namely the AutoCAD-ADS interpreter.

Any mesh generation method has to rely on a key condition that assures the generation of the mesh in any general geometry. Methods that attempt to create this condition simultaneously with the generation of the mesh cannot assure that any geometry will be discretized with the type of element desired. In this context, regarding the 3D-meshes, the automatic generation of all hexahedral meshes is still a goal to be reached. The method proposed in *plastering* is capable of meshing only simple geometries, in which intersections and closures can be solved with hexahedral elements. If more complex geometric configurations are to be discretized, the mesh cannot be generated without using some pentahedrons and pyramids, or even some tetrahedrons. This procedure can generate meshes with elements very much distorted, especially if different densities are being used. In the light of these conclusions, we propose the use of grid superposition to tackle this problem. Despite the use of some non-hexahedral elements, necessary to complete the mesh, the method seems to provide good control of their occurrence in the mesh. We also propose the association of grid superposition with the first layer of elements generated with *plastering* in order to create only hexahedral elements along the boundaries, which is strongly desired in any finite element analysis.

References

[1] K. J. Bathe. *Finite Element Procedures.* Prentice Hall, New Jersey, 1996.

[2] N. S. Lee and K. J. Bathe. Effects of Element Distortions on the Performance of Isoparametric Elements. *International Journal for Numerical Methods in Engineering*, 36:3553-3576, 1993.

[3] L. N. Gifford. More on Distorted Isoparametric Elements. *International Journal for Numerical Methods in Engineering*, 14:290-291, 1979.

[4] L. Bäcklund. On Isoparametric Elements. *International Journal for Numerical Methods in Engineering*, 12:731-732, 1978.

[5] J. A. Stricklin, W. S. Ho, E. Q. Richardson and W. E. Haisler. On Isoparametric vs Linear Strain Triangular Elements. *International Journal for Numerical Methods in Engineering*, 11:1041-1055, 1977.

[6] A. O. Cifuentes and A. Kalbag. A Performance Study of Tetrahedral and Hexahedral Elements in 3-D Finite Element Structural Analysis. *Finite Elements in Analysis and Design*, 12:313-318, 1992.

[7] K. L. Lin and H. J. Shaw. Two-Dimensional Orthogonal Grid Generation techniques. *Computers & Structures*, 41(4):569-583, 1991.

[8] K. H. Baldwin and H. L. Schreyer. Automatic Generation of Quadrilateral Elements by Conformed Mapping. *Engineering Computations*, 2:187-194, 1985.

[9] M. Montgomery and S. Fleeter. A Locally Analytic Technique Applied to Grid Generation by Elliptic Equations. *International Journal for Numerical Methods in Engineering*, 38:421-432, 1995.

[10] W. D. Rolph III. Requirements for Finite Element Model Generation from CAD Data - An Approach Using Numerical Conformal Mapping. *Computers & Structures*, 56(2/3):515-522, 1995.

[11] F. Landertshamer and H. Steffan. Method to Generate Complex Computational Meshes Efficiently. *Communications in Numerical Methods in Engineering*, 10:373-384, 1994.

[12] R. E, Smith and L. E. Eriksson. Algebraic Grid Generation. *Computer Methods in Applied Mechanics and Engineering*, 64:285-300, 1987.

[13] J. F. Thompson. A General Three-Dimensional Elliptic Grid Generation System on a Composite Block Structure. *Computer Methods in Applied Mechanics and Engineering*, 64:377-411, 1987.

[14] J. Zhu. A Hybrid Differential-Algebraic Method for Three-Dimensional Grid Generation. *International Journal for Numerical Methods in Engineering*, 29:1271-1279, 1990.

[15] P. A. F. Martins and M. J. M. B. Marques. MODEL3 - A Three-Dimensional Mesh Generator. *Computers & Structures*, 42(2):511-529, 1992.

[16] T. K. H. Tam. Finite Element Mesh Control by Integer Programming. *International Journal for Numerical Methods in Engineering*, 36:2581-2605, 1993.

[17] R. Haber, M. S. Shephard, J. F. Abel, R. H. Gallagher and D. P. Greenberg. A General Two-Dimensional, Graphical Finite Element Preprocessor Utilizing Discrete Transfinite Mappings. *International Journal for Numerical Methods in Engineering*, 17:1015-1044, 1981.

[18] T. K. H. Tam and C. G. Armstrong. 2D Finite Element Mesh Generation by Medial Axis Subdivision. *Advances in Engineering Software*, 13(5/6):313-324, 1991.

[19] T. D. Blacker, M. D. Stephenson, J. L. Mitchiner, L. R. Phillips and Y. T. Lin. Automated Quadrilateral Mesh Generation: An Knowledge System Approach. *ASME*, 88-WA/CIE-4, 1988.

[20] L. T. Souza and M. Gattass. A New Scheme for Mesh Generation and Mesh Refinement Using Graph Theory. *Computers & Structures*, 46(6):1073-1084, 1993.

[21] J. Z. Zhu, O. C. Zienkiewicz, E. Hinton and J. Wu. A New Approach to the Development of Automatic Quadrilateral Mesh Generation. *International Journal for Numerical Methods in Engineering*, 32:849-866, 1991.

[22] J. L. M. Fernandes. On Finite Element Mesh Reconstruction from Nodal Coordinates. *Advances in Engineering Software*, 17:21-27, 1993.

[23] B. P. Johnston, J. M. Sullivan Jr., A. Kwasnik. Automatic Conversion of Triangular Finite Element Meshes to Quadrilateral Elements. *International Journal for Numerical Methods in Engineering*, 31:67-84, 1991.

[24] E. Rank, M. Schweingruber and M. Sommer. Adaptive Mesh Generation and Transformation of Triangular to Quadrilateral Elements. *Communications in Numerical Methods in Engineering*, 9:121-129, 1993.

[25] G. Xie and J. A. H. Ramaekers. Graded Mesh Generation and Transformation. *Finite Element in Analysis and Design*, 17:41-55, 1993.

[26] J. M. Tembulkar and B. W. Hanks. On Generating Quadrilateral Elements from a Triangular Mesh. *Computer & Structures*, 42(4):665-667, 1992.

[27] T. D. Blacker and M. B. Stephenson. Paving: A New Approach to Automated Quadrilateral Mesh Generation. *International Journal for Numerical Methods in Engineering*, 32:811-847, 1991.

[28] T. D. Blacker, J. Jung and W. R. Witkowski. An Adaptive Finite Element Technique Using Element Equilibrium and Paving. *ASME*, 90-WA/CIE-2, 1990.

[29] T. D. Blacker, M. B. Stephenson and S. Canann. Analysis Automation with Paving: A New Quadrilateral Meshing Technique. *Advances in Engineering Software*, 13(5/6):332-337, 1991.

[30] T. D. Blacker and M. B. Stephenson. Paving: A New Approach to Automatic Quadrilateral Mesh Generation. *Sandia National Laboratories*, SAND-90-0249, 1990.

[31] T. D. Blacker and R. J. Meyers. Seams and Wedges in Plastering: A 3-D Hexahedral Mesh Generation Algorithm. *Engineering with Computers*, 9:83-93, 1993.

[32] M. B. Stephenson, S. A. Canann and T. D. Blacker. Plastering: A New Approach to Automated, 3D Hexahedral Mesh Generation. *Sandia National Laboratories*, SAND-89-2192, 1992.

[33] W. R. Buell and B. A. Bush. Mesh Generation - A Survey. *ASME*, 72-WA/DE-2, 1972.

[34] V. N. Kaliakin. A Simple Coordinate Determination Scheme for Two-Dimensional Mesh Generation. *Computers & Structures*, 43(3):505-515, 1992.

[35] S. H. Lo. Generating Quadrilateral Elements on Plane and over Curved Surfaces. *Computers & Structures*, 31(3):421-426, 1989.

[36] J. A. Talbert and A. R. Parkinson. Development of an Automatic, Two-Dimensional Finite Element Mesh Generator Using Quadrilateral Elements and Bezier Curve Boundary Definition. *International Journal for Numerical Methods in Engineering*, 29:1551-1567, 1990.

[37] C. S. Krishnamoorthy, B. Raphael and S. Mukherjee. Meshing by Successive Superelement Decomposition (MSD) - A New Approach to Quadrilateral Mesh Generation. *Finite Element in Analysis and Design*, 20:1-37, 1995.

[38] J. F. Thompson, Z. U. A. Warsi and C. W. Mastin. *Numerical Grid Generation - Foundations and Applications*. North-Holland, New York, 1985.

[39] P. L. George. *Automatic Mesh Generation - Application to Finite Element Methods.* John Wiley & Sons, Masson, 1991.

[40] P. Knupp and S. Steinberg. *Fundamentals of Grid Generation.* CRC Press, Boca Raton, 1994.

[41] A. Tezuka. Adaptive Process with Quadrilateral Finite Elements. *Advances in Engineering Software*, 15:185-201, 1992.

[42] P. L. Baehmann, S. L. Wittchen, M. S. Shephard, K. R. Grice and M. A. Yerry. Robust, Geometrically Based, Automatic Two-Dimensional Mesh Generation. *International Journal for Numerical Methods in Engineering*, 24:1043-1078, 1987.

[43] F. Cheng, J. W. Jaromczyk, J. R. Lin, S. S. Chang and J. Y. Lu. A Parallel Mesh Generation Algorithm Based on Vertex Label Assignment Scheme. *International Journal for Numerical Methods in Engineering*, 28:1429-1448, 1989.

[44] A. S. Watson and C. J. Anumba. The Need for an Integrated 2D/3D CAD System in Structural Engineering. *Computers & Structures*, 41(6):1175-1182, 1991.

[45] H. Jiazhen. Intoad - An Interface Between IRM and ADINA. *Computers & Structures*, 47(4/5):751-756, 1993.

[46] J. Hoyte. The Cut & Glue Mesh Generation Algorithm. *Engineering with Computers*, 8:51-58, 1992.

[47] S. H. Lo. Volume Discretization into Tetrahedra - I. Verification and Orientation of Boundary Surfaces. *Computers & Structures*, 39(5):493-500, 1991.

[48] S. H. Lo. Volume Discretization into Tetrahedra - II. 3D Triangulation by Advancing Front Approach. *Computers & Structures*, 39(5):501-511, 1991.

[49] B. P. Johnston. A Normal Offsetting Technique for Automatic Mesh Generation in Three Dimensions. *International Journal for Numerical Methods in Engineering*, 36:1717-1735, 1993.

[50] A. Kela, M. Saxena and R. Perucchio. A Hierarchical Structure for Automatic Meshing and Adaptive FEM Analysis. *Engineering Computations*, 4:104-112, 1987.

[51] M. Saxena and R. Perucchio. Element Extraction for Automatic Meshing Based on Recursive Spatial Decompositions. *Computers & Structures*, 36(3):513-529, 1990.

[52] R. Perucchio, M. Saxena and A. Kela. Automatic Mesh Generation from Solid Models Based on Recursive Spatial Decompositions. *International Journal for Numerical Methods in Engineering*, 28:2469-2501, 1989.

[53] M. S. Shephard and M. K. Georges. Automatic Three-Dimensional Mesh Generation by the Finite Octree Technique. *International Journal for Numerical Methods in Engineering*, 32:709-749, 1991.

[54] E. K. Buratynski. A Fully Automatic Three-Dimensional Mesh Generator for Complex Geometries. *International Journal for Numerical Methods in Engineering*, 30:931-952, 1990.

[55] J. E. Castillo and S. Steinberg. On the Folding of Numerical Generated Grids: Use of a Reference Grid. *Communications in Applied Numerical Methods* , 4:471-481, 1988.

[56] L. Sezer and I. Zeid. Automatic Quadrilateral/Triangular Free-Form Mesh Generation for Planar Regions. *International Journal for Numerical Methods in Engineering*, 32:1441-1483, 1991.

[57] K. J. Berry. Parametric 3D Finite-Element Mesh Generation. *Computers & Structures*, 33(4):969-976, 1989.

[58] R. Perucchio, A. R. Ingraffea and J. F. Abel. Interactive Computer Graphic Preprocessing for Three-Dimensional Finite Element Analysis. *International Journal for Numerical Methods in Engineering*, 18:909-926, 1982.

[59] A. Oddy, J. Goldak, M. McDill and M. Bibby. A Distortion Metric for Isoparametric Finite Elements. *Transactions of the CSME*, 12(4):213-217, 1988.

[60] S. A. Canann, M. B. Stephenson and T. Blacker. Optsmoothing: An Optimization-Driven Approach to Mesh Smoothing. *Finite Elements in Analysis and Design*, 13:185-190, 1993.

[61] M. Sabin. Criteria for Comparison of Automatic Mesh Generation Methods. *Advances in Engineering Software*, 13(5/6):220-225, 1991.

[62] E. Amezua, M. V. Hormaza, A. Hernández and M. B. G. Ajuria. A Method for the Improvement of 3D Solid Finite-Element Meshes. *Advances in Engineering Software*, 22(1):45-53, 1995.

[63] J. Robinson. Some New Distortion Measures for Quadrilaterals. *Finite Elements in Analysis and Design*, 3:183-197, 1987.

[64] S. E. Benzley, K. Merkley, T. D. Blacker and L. Schoof. Pre- and Post-Processing for the Finite Element Method. *Finite Elements in Analysis and Design*, 19:243-260, 1995.

[65] W. C. Thacker. A Brief Review of Techniques for Generating Irregular Computational Grids. *International Journal for Numerical Methods in Engineering*, 15:1335-1341, 1980.

[66] K. H. Le. Finite Element Mesh Generation Methods: A Review and Classification. *Computer-Aided design*, 20(1):27-38, 1889.

Appendix A

Code for quadrilateral and hexahedral mesh generation

A.1 Main body of the code for the 2D and 3D-mesh generation

```
#include <stdio.h>
#include <math.h>
#include <adslib.h>
#include <aplib.h>
#include <time.h>
#include "global_variables.h"
#include "auxiliary_functions.c"
#include "nodes_and_elements_on_edges_of_solid.c"
#include "initial_2D_front.c"
#include "orientation.c"
#include "front_angle_distance.c"
#include "close_corner.c"
#include "seam.c"
#include "auxiliary_functions_for_advance_front.c"
#include "advance_front.c"
#include "auxiliary_functions_for_correct_front_size.c"
#include "correct_front_size.c"
#include "auxiliary_functions_for_intersection.c"
#include "intersection.c"
#include "close_front.c"
#include "smooth_front.c"
#include "cleanp.c"
#include "smooth.c"
#include "initial_3D_front.c"
#include "face_angles.c"
#include "auxiliary_functions_for_checking_mesh.c"
#include "auxiliary_functions_for_advance_face_front.c"
#include "advance_face_front.c"
#include "smooth_3D.c"
#include "template.c"

/*#####################*/
/* GEOMETRY EXTRACTION */
/*#####################*/
geo()
{

/* DECLARATION OF LOCAL VARIABLES */
  int i, j;
  ads_name sol_set, sol_name;
  REAL aux;
  time_t ti, tf;
  ap_Edgelist *sol_elist, *face_elist;
  ap_Facelist *sol_flist;

/* INITIAL SETTINGS */
  ti=time(&ti);
  a45  = acos(-1.0) / 4.0 ;
  a60  = 4.0 * a45 / 3.0;
  a90  = 2.0 * a45;
  a120 = 2.0 * a60;
  a135 = 3.0 * a45;
  a180 = 4.0 * a45;
  a200 = 5.0 * a120 / 3.0;
  a220 = 11.0 * a120 / 6.0;
  a240 = 2.0 * a120;
  a300 = 5.0 * a60;
  for(i=0 ; i < MAX_NODE ; i++) node[i].nfi = 0;
  for(i=0 ; i < 6*MAX_FRONT_PER_FACE ; i++) no_intersect_node[i] = 0;
  ads_command(RTSTR, "setvar", RTSTR, "CMDECHO" ,RTSTR, "0",  0);
  ads_command(RTSTR, "handles", RTSTR, "on",  0);

/* SOLID SELECTION AND ID */
  ads_ssget(NULL, NULL, NULL, NULL, sol_set);
  ads_ssname(sol_set, 0L, sol_name);
  ap_name2obj(sol_name, &sol_id);

/* INPUT ELEMENT SIZE */
  ads_getreal("Enter element size (units of drawing):  ", &size);

/* MESH'S CONTRUCTION VIZUALIZATION  */
  ads_getint("Do you want to see the mesh construction ?  (1, 0) ", &repr);

/* SOLID's LIST OF EDGES */
  printf("Stracting edges geometry...\n");
  ap_obj2edges(sol_id, TRUE, &sol_elist);
  while(sol_elist){
    sol_edge[n_edge].id = sol_elist->edge_id;
    sol_edge[n_edge].len = (REAL) (sol_elist->edge->edge_len);
    sol_edge[n_edge].s_par = (REAL) (sol_elist->edge->s_parm);
    sol_edge[n_edge].e_par = (REAL) (sol_elist->edge->e_parm);
    sol_edge[n_edge].s_p[X] = (REAL) (sol_elist->edge->s_pt[X]);
    sol_edge[n_edge].s_p[Y] = (REAL) (sol_elist->edge->s_pt[Y]);
    sol_edge[n_edge].s_p[Z] = (REAL) (sol_elist->edge->s_pt[Z]);
    sol_edge[n_edge].e_p[X] = (REAL) (sol_elist->edge->e_pt[X]);
    sol_edge[n_edge].e_p[Y] = (REAL) (sol_elist->edge->e_pt[Y]);
    sol_edge[n_edge].e_p[Z] = (REAL) (sol_elist->edge->e_pt[Z]);
    sol_elist = sol_elist->edgenext;
    n_edge++;
  }
  ap_free_edgelist(sol_elist);
  set_edge_size();

/* SOLID's LIST OF FACES */
  printf("Stracting faces geometry...\n");
  ap_obj2faces(sol_id, TRUE, &sol_flist);
  while(sol_flist){
    sol_face[n_face].id = sol_flist->face_id;
    sol_face[n_face].type = sol_flist->face->stype;
    ap_face2edges(sol_id, sol_face[n_face].id, FALSE, &face_elist);
    sol_face[n_face].ne = 0;
    while(face_elist){
      for(i=0 ; i < n_edge ; i++){
        if(sol_edge[i].id == face_elist->edge_id){
          sol_face[n_face].edge[sol_face[n_face].ne] = &sol_edge[i];
          break;
        }
      }
      face_elist = face_elist->edgenext;
      sol_face[n_face].ne++;
    }
    sol_flist = sol_flist->facenext;
    n_face++;
  }
  ap_free_edgelist(face_elist);
  ap_free_facelist(sol_flist);

/* CREATE NODES AND ELEMENTS IN EACH SOLID'S EDGE */
  printf("Creating nodes and elements on the edges...\n");
  nodes_and_elements_on_edges_of_solid();

/* SETUP OF 2D FRONT */
  printf("Setting up 2D-fronts...\n");
  for(m=0 ; m < n_face ; m++){
    initial_2D_front();
    for(j=0 ; j <= n_f[m] ; j++){
      orientation(j);
      printf("turn[%d][%d]=%2.1lf\n", m, j, turn[m][j]);
      front_angle_distance(j);
    }
    set_edge_size_of_face();
  }

/* TIME MEASUREMENT */
  tf=time(&tf);
  aux = difftime(tf, ti);
  printf("CPU time to get the geometry = %f sec.\n", aux);
  ti = tf;
  return RSRSLT;
}

/*#########*/
/* 2D-MESH */
/*#########*/
int autoqm()
{
  int i, j, k, e, s, c=0;
  REAL aux;
  time_t ti, tf;

/* MESH EACH FACE */
  ti=time(&ti);
  for(m=0 ; m < n_face; m++){
    if(sol_face[m].type == 0)
      ap_pt_norm2face(sol_id, sol_face[m].id,node[f[m][0].n].coord,TRUE, n[m]);
    printf("\nface = %d\n", m);
    printf("   rect_face=%d\n", (rect_face = rectangular_face(m)));
```

```
pf = fi;
pn = ni;
for(j=0 ; j < MAX_FRONT_PER_FACE ; j++){
  origin[j].front[0] = j;
  origin[j].nfri = 1;
}
for(j=0 ; j <= n_f[m] ; j++){pff[j] = fi; printf("%d\n",advance_front(j));}

for(j=0 ; j <= n_f[m] ; j++){
  printf(" front = %d\n", j);
  if(set_ki_kpi(j) == 0) continue;
  for( ; ; ){
    if((kpi-ki+1) <= 6){close_front(j); break;}
    e = 0;
    s = 0;
    while(seam(j) == 1){
      s = 1;
      printf("1\n");
      if((kpi-ki+1) <= 6){
        close_front(j);
        e = 1;
        break;
      }
    }
    if(s == 1) if(repr == 1) reprint_face(pff[j]);
    if(e == 1) break;
    printf("0\n");
    e = 0;
    while(close_corner(j) == 1){
      printf("1\n");
      if((kpi-ki+1) <= 6){
        close_front(j);
        e = 1;
        break;
      }
    }
    if(e == 1) break;
    printf("0\n");
    e = 0;
    for(i=0 ; i <= n_f[m] ; i++){
      if((j == i) || (check_origin(i, j) == 0)){
        if(intersection(j, i) == 1){
          printf("1\n");
          if(j <= i) j--; else j = i - 1;
          e = 1;
          break;
        }
        printf("0\n");
      }
    }
    if(e == 1) break;
    if(smooth_front(j) == 1){
      printf("1\n");
      e = 0;
```

```
          for(i=0 ; i <= n_f[m] ; i++){
            if((j == i) || (check_origin(i, j) == 0)){
              if(intersection(j, i) == 1){
                printf("1\n");
                if(j <= i) j--; else j = i - 1;
                e = 1;
                break;
              }
              printf("0\n");
            }
          }
          if(e == 1) break;
        }
        else printf("0\n");
        pfc = pff[j];
        pff[j] = fi;
        printf("%d\n", advance_front(j));
        if(correct_front_size(j) == 1){
          printf("1\n");
          printf("%d\n", smooth_front(j));
        }
        else printf("0\n");
      }
    }
    smooth();
    clean_up();
  }

/* TIME MEASUREMENT */
  tf=time(&tf);
  aux = difftime(tf, ti);
  printf("CPU time for meshing 2D = %f sec.\n", aux);

  if(repr == 0) print_all_faces();
  for(i=0 ; i < MAX_SOL_FACE ; i++) free(f[i]);
  free(f);
  free(af);
  free(edge);
  free(sol_edge);
  return RSRSLT;
}

/*#########*/
/* 3D-MESH */
/*#########*/
int mesh3D()
{

/* DECLARATION OF LOCAL VARIABLES */
  int i, k;
  REAL vol, aux;
  time_t ti, tf;

/* MESHING EACH FACE */
```

```
  ti=time(&ti);
  pn = ni;
  initial_3D_front();
  for(i=0 ; i < fi ; i++){
    if(face[i].no != -1){
      face[i].ord = 0;
      for(k=0 ; k < 4 ; k++){
        face[i].a[k] = face_angles(i, k);
      }
    }
  }
  for(i=0 ; i < MAX_NODE ; i++) node[i].type = -1;
  node[0].type = 1;
  set_node_type_face();
  vol = volume_of_2D_mesh(0);
  i = 0;
  while(advance_face_front(i) > 0) i++;
  check_two_elements_per_face();
  smooth_3D();
  set_node_type_element();
  printf("Volume of 2D-mesh = %lf\n", vol);
  printf("Volume of 3D-mesh = %lf\n",(aux=volume_of_3D_mesh(0,ci)));
  printf("Difference:  volume 2D-mesh - volume 3D-mesh = %lf\n",(aux-vol));

/* TIME MEASUREMENT */
  tf=time(&tf);
  aux = difftime(tf, ti);
  printf("CPU time for meshing 3D = %f sec.\n", aux);

  return RSRSLT;
}
```

A.2 Declaration of global variables

```
#define CUBE 30
#define MAX_SOL_FACE CUBE
#define MAX_ELEMENT CUBE*CUBE*CUBE
#define MAX_FACE 2*(CUBE+1)*(CUBE+1)*(CUBE+1)
#define MAX_NODE (CUBE+1)*(CUBE+1)*(CUBE+1)
#define MAX_FRONT CUBE*CUBE*CUBE
#define MAX_FRONT_PER_FACE 2*CUBE
#define MAX_EDGE CUBE*CUBE*CUBE
typedef double REAL;
typedef REAL POINT[3];

/* GEOMETRY DATA */
struct solid_edge
{
  long int id;
  int nel;
  REAL len;
  REAL s_par, e_par;
  POINT s_p, e_p;
  int ref;
  REAL size;
}sol_edge[3*MAX_SOL_FACE];

struct solid_face
{
  int id;
  int type;
  int ne;
  struct solid_edge *edge[CUBE];
}sol_face[MAX_SOL_FACE];

/* 2D AND 3D DATABASES */
struct element
{
  int no;
  int enode[8];
}element[MAX_ELEMENT];

struct face
{
  int no;
  int fm;
  int fnode[4];
  int ord;
  int fface[4];
  REAL a[4];
  POINT norm;
}face[MAX_FACE];

struct edge
{
  int no;
  int node[2];
  int edge_id;
}edge[MAX_EDGE];

struct node
{
  int no;
  POINT coord;
  int nface[15];
  int nfi;
  int type;
}node[MAX_NODE];

/* 2D FRONT STRUCTURE */
struct front
{
  int n;
  int f;
  REAL a;
  REAL d;
}f[MAX_SOL_FACE][MAX_FRONT];

/* DECLARATION OF GLOBAL VARIABLES */
int ni = 0;                                                          /* NODES COUNTER */
int ei = 0;                                                          /* EDGES COUNTER */
int vi = 0;                                                         /* VERTEX COUNTER */
int fi = 0;                                                          /* FACES COUNTER */
int ci = 0;                                                       /* ELEMENTS COUNTER */
int pi[MAX_FRONT_PER_FACE];                                      /* 2D-FRONT COUNTER */
int n_f[MAX_FRONT_PER_FACE];              /* No. OF FRONT IN THE FACE BEING MESHED */
int ff[MAX_FRONT];                 /* AUXILIARY ARRAY OF 2D-FRONT IN advance_front */
int tfi;                                                /* TEMPORARY COUNTER FOR fi */
int afi;                                           /* *ff COUNTER AND FACE COUNTER */
int lafi;                                                /* AUXILIARY FACE COUNTER */
int aci;                                              /* AUXILIARY ELEMENT COUNTER */
int ki, kpi, kj, kpj;                            /* LOCAL BEGIN AND END OF 2D-FRONT */
int n_edge=0, n_face=0;                     /* No. OF EDGES AND FACES IN THE SOLID */
int m;                                                 /* ACTUAL FACE BEING MESHED */
int pf=0;          /* COUNTER OF FACES IN FACE OF THE SOLID FOR FUNCTION smooth */
int pfc;                     /* COUNTER OF FACES GENERATED FOR correct_front_size */
int pn;            /* COUNTER OF NODES IN FACE OF THE SOLID FOR FUNCTION smooth */
int pff[MAX_FRONT_PER_FACE];/* COUNTER OF FACES GENERATED SINCE adance_front */
int rect_face;                                   /* COUNTER FOR RECTANGULAR FACE */
int no_intersect_node[6*MAX_FRONT_PER_FACE];     /* AUXILIARY FOR intersection */
int n_i_n=0;                                /* COUNTER FOR NO INTERSECTION NODES */
int repr;                 /* CONTROL OF PRINTING WHILE CONSTRUCTING THE MESH */
REAL turn[MAX_SOL_FACE][MAX_FRONT_PER_FACE];         /* 2D-FRONT LOOP DIRECTION */
REAL a45, a60, a90, a120, a135, a180, a200, a220, a240, a300;         /* ANGLES */
REAL size;                                                        /* ELEMENT SIZE */
REAL msize[MAX_FACE];                                /* AVERAGE OF ELEMENT SIZES */
POINT n[MAX_SOL_FACE];                                                  /* NORMAL */
ap_Objid sol_id;                                             /* SOLID ID VARIABLE */

struct origin
{
  int nfri;
  int front[MAX_FRONT_PER_FACE];
}origin[MAX_FRONT_PER_FACE];          /* CHECK OF ORIGIN OF FRONT intersection */

struct front af[MAX_FRONT];                       /* AUXILIARY ARRAY OF 2D-FRONT */
```

A.3 General auxiliary functions

```
/* PROJECT A POINT ON THE SURFACE */
void point_projection(REAL *pt1)
{
  POINT pt2;
  ap_pt2face(sol_id, sol_face[m].id, pt1, pt2);
  pt1[X] = pt2[X];
  pt1[Y] = pt2[Y];
  pt1[Z] = pt2[Z];
}

/* CREATE VECTOR FROM TWO POINTS IN PAVING FRONT */
void vector(int i0, int i1, REAL *vt)
  vt[X] = node[f[m][i1].n].coord[X] - node[f[m][i0].n].coord[X];
  vt[Y] = node[f[m][i1].n].coord[Y] - node[f[m][i0].n].coord[Y];
  vt[Z] = node[f[m][i1].n].coord[Z] - node[f[m][i0].n].coord[Z];
}

/* CREATE VECTOR FROM TWO NODAL POINTS */
void vector_no(int n0, int n1, REAL *vt)
  vt[X] = node[n1].coord[X] - node[n0].coord[X];
  vt[Y] = node[n1].coord[Y] - node[n0].coord[Y];
  vt[Z] = node[n1].coord[Z] - node[n0].coord[Z];
}

/* SUBTRACT TWO VECTOR */
void sub_vector(REAL *vt0, REAL *vt1, REAL *vt)
{
  vt[X] = vt1[X] - vt0[X];
  vt[Y] = vt1[Y] - vt0[Y];
  vt[Z] = vt1[Z] - vt0[Z];
}

/* ADD TWO VECTOR */
void add_vector(REAL *vt0, REAL *vt1, REAL *vt)
{
  vt[X] = vt1[X] + vt0[X];
  vt[Y] = vt1[Y] + vt0[Y];
  vt[Z] = vt1[Z] + vt0[Z];
}

/* VECTOR PRODUCT */
void vector_product(REAL *vt0, REAL *vt1, REAL *vt)
{
  vt[X] = (vt0[Y] * vt1[Z] - vt0[Z] * vt1[Y]);
  vt[Y] = (vt0[Z] * vt1[X] - vt0[X] * vt1[Z]);
  vt[Z] = (vt0[X] * vt1[Y] - vt0[Y] * vt1[X]);
}

/* INNER PRODUCT */
REAL inner_product(REAL *vt0, REAL *vt1)
{
  return (vt0[X] * vt1[X] + vt0[Y] * vt1[Y] + vt0[Z] * vt1[Z]);
}

/* POINT OF PAVING FRONT + VECTOR */
void point_vector(int i, REAL *vt, REAL *pt)
{
  pt[X] = vt[X] + node[f[m][i].n].coord[X];
  pt[Y] = vt[Y] + node[f[m][i].n].coord[Y];
  pt[Z] = vt[Z] + node[f[m][i].n].coord[Z];
}

/* NODE + VECTOR */
void point_vector_no(int i, REAL *vt, REAL *pt)
{
  pt[X] = vt[X] + node[i].coord[X];
  pt[Y] = vt[Y] + node[i].coord[Y];
  pt[Z] = vt[Z] + node[i].coord[Z];
}

/* SCALAR * VECTOR */
void scalar_vector(REAL sc, REAL *vt)
{
  vt[X] = sc * vt[X];
  vt[Y] = sc * vt[Y];
  vt[Z] = sc * vt[Z];
}

/* VECTOR NORM */
REAL vector_norm(REAL *vt)
{
  return (sqrt(pow(vt[X],2.0) + pow(vt[Y],2.0) + pow(vt[Z],2.0)));
}

/* INCREMENT AND DECREMENT (ki, kpi) */
int incr(int i){
  return ((i+1) + (ki - i - 1) * (i == kpi));
}
int decr(int i){
  return ((i-1) + (kpi - i + 1) * (i == ki));
}

/* INCREMENT AND DECREMENT (kj, kpj) */
int inc(int i){
  return ((i+1) + (kj - i - 1) * (i == kpj));
}
int dec(int i){
  return ((i-1) + (kpj - i + 1) * (i == kj));
}

/* INCREMENT AND DECREMENT (0, 3) */
int inc4(int i){
  return ((i+1) * (1 - (i == 3)));
}
int dec4(int i){
  return ((i-1) + (3 - i + 1) * (i == 0));
}

/* INCREMENT (0, tfi-1) */
int inct(int i){
  return ((i+1) * (1 - (i == (tfi-1))));
}
```

```
/* INCREMENT (0, ni-1) */
int incn(int i){
  return ((i+1) * (1 - (i == (ni-1))));
}

/* SET ELEMENT SIZE OF SOLID'S EDGES */
void set_edge_size()
{
  int i, j, w=0;
  long int ident, id[CUBE];
  REAL aux, aux1, si[CUBE];
  ads_getint("Different local size ?  (1, 0) ", &i);
  if(i == 1){
    for( ; ; ){
      ap_sel_edge("Pick the edge\n", &sol_id, &ident);
      id[w] = ident;
      ads_getreal("Enter edge's density (units of drawing):  ", &aux);
      si[w] = aux;
      w++;
      ads_getint("Another edge ?  (1, 0) ", &i);
      if(i == 0) break;
    }
  }
  for(i=0 ; i < n_edge ; i++){
    sol_edge[i].size = size;
    sol_edge[i].ref = 0;
    for(j=0 ; j < w ; j++){
      if(sol_edge[i].id == id[j]){
        sol_edge[i].size = si[j];
        sol_edge[i].ref = 1;
      }
    }
    aux = sol_edge[i].len / sol_edge[i].size;
    if(aux < 1.5){
      sol_edge[i].nel = 2;
      sol_edge[i].ref = 1;
    }
    else{
      aux1 = aux - floor(aux);
      if(aux1 < 0.5) sol_edge[i].nel = (int) floor(aux);
      else sol_edge[i].nel = (int) ceil(aux);
      if(sol_edge[i].nel & 1) sol_edge[i].nel++;
    }
  }
}

/* SET MINIMUM SIZE OF EDGE IN EACH FACE SIZE */
void set_edge_size_of_face()
{
  int i, w=0;
  msize[m] = 0.0;
  for(i=0 ; i <= pi[m] ; i++){
    msize[m] = msize[m] + f[m][i].d;
    w++;
  }
  msize[m] = msize[m] / (REAL) w;
}

/* CREATE A FACE FROM ONE KNOWN EDGES */
void create_face(int n1, int n2, int n3, int n4)
{
  face[fi].no = fi;
  face[fi].fnode[0] = n1;
  face[fi].fnode[1] = n2;
  face[fi].fnode[2] = n3;
  face[fi].fnode[3] = n4;
  node[n1].nface[node[n1].nfi] = fi;
  node[n1].nfi++;
  node[n2].nface[node[n2].nfi] = fi;
  node[n2].nfi++;
  node[n3].nface[node[n3].nfi] = fi;
  node[n3].nfi++;
  node[n4].nface[node[n4].nfi] = fi;
  node[n4].nfi++;
  face[fi].fm = m;
  fi++;
  if(repr == 1) ads_command(RTSTR, "3dface", RT3DPOINT,
                         node[face[fi-1].fnode[0]].coord,
                         RT3DPOINT, node[face[fi-1].fnode[1]].coord,
                         RT3DPOINT, node[face[fi-1].fnode[2]].coord,
                         RT3DPOINT, node[face[fi-1].fnode[3]].coord,
                         RTSTR, "", 0);
}

/* DISTANCE BETWEEN TWO POINTS */
REAL dist(REAL *p, REAL *r)
{
  return (sqrt(pow(p[X]-r[X],2.0) + pow(p[Y]-r[Y],2.0) + pow(p[Z]-r[Z],2.0)));
}

/* SET ki AND kpi */
int set_ki_kpi(int nf)
{
  int i, j;
  ki = 0;
  kpi = 0;
  for(i=0 ; i <= pi[m] ; i++){
    if(f[m][i].f == nf){
      ki = i;
      i++;
      for(j=i ; j <= pi[m] ; j++) if(f[m][j].f == nf) { kpi = j;  break;}
      break;
    }
  }
  return (ki+kpi);
}

/* REPRINT FACES */
```

```
void reprint_face(int f)
{
  int i;
  printf("reprint_face  f=%d  fi=%d\n", f, fi);
  for(i=f ; i < fi ; i++){
    if(face[i].no != -1)
      ads_command(RTSTR, "erase", RTSTR, "l", RTSTR, "", 0);
  }
  for(i=f ; i < fi ; i++){
    if(face[i].no != -1)
       ads_command(RTSTR, "3dface", RT3DPOINT, node[face[i].fnode[0]].coord,
                   RT3DPOINT, node[face[i].fnode[1]].coord,
                   RT3DPOINT, node[face[i].fnode[2]].coord,
                   RT3DPOINT, node[face[i].fnode[3]].coord,
                   RTSTR, "", 0);
  }
}

/* PRINT ALL FACES */
void print_all_faces()
{
  int i;
  for(i=0 ; i < fi ; i++){
    if(face[i].no != -1)
       ads_command(RTSTR, "3dface", RT3DPOINT, node[face[i].fnode[0]].coord,
                   RT3DPOINT, node[face[i].fnode[1]].coord,
                   RT3DPOINT, node[face[i].fnode[2]].coord,
                   RT3DPOINT, node[face[i].fnode[3]].coord,
                   RTSTR, "", 0);
  }
}

/* RECTANGULAR FACE */
int rectangular_face(int f)
{
  int i, j;
  REAL d1, d2, tol=0.1;
  POINT P1, P2, P3, P4;
  if(sol_face[f].ne == 4){
    P1[X] = sol_face[f].edge[0]->s_p[X];
    P1[Y] = sol_face[f].edge[0]->s_p[Y];
    P1[Z] = sol_face[f].edge[0]->s_p[Z];
    P2[X] = sol_face[f].edge[0]->e_p[X];
    P2[Y] = sol_face[f].edge[0]->e_p[Y];
    P2[Z] = sol_face[f].edge[0]->e_p[Z];
    for(i=1 ; i < 4 ; i++){
      if(dist(P2, sol_face[f].edge[i]->s_p) < tol){
        P3[X] = sol_face[f].edge[i]->e_p[X];
        P3[Y] = sol_face[f].edge[i]->e_p[Y];
        P3[Z] = sol_face[f].edge[i]->e_p[Z];
        j = i;
        break;
      }
      if(dist(P2, sol_face[f].edge[i]->e_p) < tol){
        P3[X] = sol_face[f].edge[i]->s_p[X];
        P3[Y] = sol_face[f].edge[i]->s_p[Y];
        P3[Z] = sol_face[f].edge[i]->s_p[Z];
        j = i;
        break;
      }
    }
    for(i=1 ; i < 4 ; i++){
      if(i != j){
        if(dist(P3, sol_face[f].edge[i]->s_p) < tol){
          P4[X] = sol_face[f].edge[i]->e_p[X];
          P4[Y] = sol_face[f].edge[i]->e_p[Y];
          P4[Z] = sol_face[f].edge[i]->e_p[Z];
          break;
        }
        if(dist(P3, sol_face[f].edge[i]->e_p) < tol){
          P4[X] = sol_face[f].edge[i]->s_p[X];
          P4[Y] = sol_face[f].edge[i]->s_p[Y];
          P4[Z] = sol_face[f].edge[i]->s_p[Z];
          break;
        }
      }
    }
    d1 = dist(P1, P3);
    d2 = dist(P2, P4);
    if((fabs(dist(P1,P2) - dist(P3, P4)) < tol) &&
       (fabs(dist(P1,P4) - dist(P2, P3)) < tol) &&
       (fabs(d1-d2) < tol)) return 1;
  }
  return 0;
}

/* COMMUTE FACE NODES */
void commute_face_nodes(int i)
{
  int w;
  if(node[i].nfi == 2){
    w = face[node[i].nface[1]].fnode[0];
    if(node[face[node[i].nface[1]].fnode[1]].nfi == 2){
      face[node[i].nface[1]].fnode[0] = face[node[i].nface[1]].fnode[1];
      face[node[i].nface[1]].fnode[1] = w;
      w = face[node[i].nface[1]].fnode[2];
      face[node[i].nface[1]].fnode[2] = face[node[i].nface[1]].fnode[3];
      face[node[i].nface[1]].fnode[3] = w;
    }
    else{
      face[node[i].nface[1]].fnode[0] = face[node[i].nface[1]].fnode[2];
      face[node[i].nface[1]].fnode[2] = w;
    }

    w = node[i].nface[1];
    node[i].nface[1] = node[i].nface[0];
    node[i].nface[0] = w;
  }
```

```
}

/* SET ORIGIN */
void set_origin(int nff, int nf)
{
  int i;
  for(i=0 ; i < origin[nf].nfri ; i++){
    origin[nff].front[origin[nff].nfri] = origin[nf].front[i];
    origin[nff].nfri++;
  }
}

/* CHECK ORIGIN */
int check_origin(int nff, int nf)
{
  int i, j;
  for(i=0 ; i < origin[nf].nfri ; i++){
    for(j=0 ; j < origin[nff].nfri ; j++){
      if(origin[nf].front[i] == origin[nff].front[j]) return 1;
    }
  }
  return 0;
}

/* CREATE NORMAL VECTOR OF FACE */
void face_norm(int n0, int n1, int n2, int n3, REAL *vn)
{
  REAL d;
  POINT v0, v1;
  vector_no(n0, n2, v0);
  vector_no(n1, n3, v1);
  vector_product(v0, v1, vn);
  d = vector_norm(vn);
  scalar_vector(1.0/d, vn);
}

/* REPRINT ELEMENTS */
void reprint_element(int s)
{
  int i, k;
  printf("reprint_element  from=%d to ci=%d\n", s, ci);
  for(i=s ; i < ci ; i++)
    for(k=0 ; k < 6 ; k++)
      ads_command(RTSTR, "erase", RTSTR, "l", RTSTR, "", 0);
  for(i=s ; i < ci ; i++){
    ads_command(RTSTR, "3dface", RT3DPOINT, node[element[i].enode[0]].coord,
               RT3DPOINT, node[element[i].enode[1]].coord,
               RT3DPOINT, node[element[i].enode[2]].coord,
               RT3DPOINT, node[element[i].enode[3]].coord,
               RTSTR, "", 0);
    ads_command(RTSTR, "3dface", RT3DPOINT, node[element[i].enode[0]].coord,
               RT3DPOINT, node[element[i].enode[1]].coord,
               RT3DPOINT, node[element[i].enode[5]].coord,
               RT3DPOINT, node[element[i].enode[4]].coord,
                RTSTR, "", 0);
    ads_command(RTSTR, "3dface", RT3DPOINT, node[element[i].enode[1]].coord,
                RT3DPOINT, node[element[i].enode[2]].coord,
                RT3DPOINT, node[element[i].enode[6]].coord,
                RT3DPOINT, node[element[i].enode[5]].coord,
                RTSTR, "", 0);
    ads_command(RTSTR, "3dface", RT3DPOINT, node[element[i].enode[2]].coord,
                RT3DPOINT, node[element[i].enode[3]].coord,
                RT3DPOINT, node[element[i].enode[7]].coord,
                RT3DPOINT, node[element[i].enode[6]].coord,
                RTSTR, "", 0);
    ads_command(RTSTR, "3dface", RT3DPOINT, node[element[i].enode[3]].coord,
                RT3DPOINT, node[element[i].enode[0]].coord,
                RT3DPOINT, node[element[i].enode[4]].coord,
                RT3DPOINT, node[element[i].enode[7]].coord,
                RTSTR, "", 0);
    ads_command(RTSTR, "3dface", RT3DPOINT, node[element[i].enode[4]].coord,
                RT3DPOINT, node[element[i].enode[5]].coord,
                RT3DPOINT, node[element[i].enode[6]].coord,
                RT3DPOINT, node[element[i].enode[7]].coord,
                RTSTR, "", 0);
  }
}
```

A.4 Function *nodes_and_elements_on_edges_of_solid()*

```
void nodes_and_elements_on_edges_of_solid()
{
  int i, j, k, ii, s, e, sd, ed;
  REAL t, dt, tol = 0.01, ratio, aux, tx, rt, es, ss;
  for(ii=0 ; ii < n_edge ; ii++){
    s = 0;
    e = 0;
    for(i=0 ; i < ni ; i++){
      if(dist(node[i].coord, sol_edge[ii].s_p) < tol){
        s = 1;
        break;
      }
    }
    for(i=0 ; i < ni ; i++){
      if(dist(node[i].coord, sol_edge[ii].e_p) < tol){
        e = 1;
        break;
      }
    }
    if(dist(sol_edge[ii].e_p, sol_edge[ii].s_p) < tol) e = 1;
    if(s == 0){
      node[ni].coord[X] = sol_edge[ii].s_p[X];
      node[ni].coord[Y] = sol_edge[ii].s_p[Y];
      node[ni].coord[Z] = sol_edge[ii].s_p[Z];
      node[ni].no = ni;
      ni++;
      vi++;
    }
    if(e == 0){
      node[ni].coord[X] = sol_edge[ii].e_p[X];
      node[ni].coord[Y] = sol_edge[ii].e_p[Y];
      node[ni].coord[Z] = sol_edge[ii].e_p[Z];
      node[ni].no = ni;
      ni++;
      vi++;
    }
  }
  for(ii=0 ; ii < n_edge ; ii++){
    for(i=0 ; i < vi ; i++){
      if(dist(node[i].coord, sol_edge[ii].s_p) < tol){
        edge[ei].node[0] = i;
        edge[ei].no = ei;
        edge[ei].edge_id = sol_edge[ii].id;
        break;
      }
    }
    k = sol_edge[ii].nel;
    ratio = 0.0;
    rt = 0.0;
    dt = (sol_edge[ii].e_par − sol_edge[ii].s_par) / (REAL) sol_edge[ii].nel;
    t = sol_edge[ii].s_par + dt;
    if(sol_edge[ii].ref == 0){
      ss = size;
      es = size;
      sd = −1;
      ed = −1;
      for(j=0 ; j < n_edge ; j++){
        if(sol_edge[j].ref == 1){
          if((dist(sol_edge[ii].s_p, sol_edge[j].s_p) < tol) ||
              (dist(sol_edge[ii].s_p, sol_edge[j].e_p) < tol)) sd = j;
          if((dist(sol_edge[ii].e_p, sol_edge[j].s_p) < tol) ||
              (dist(sol_edge[ii].e_p, sol_edge[j].e_p) < tol)) ed = j;
        }
      }
      if(ed != sd){
        if(sd != −1) ss = sol_edge[sd].size;
        if(ed != −1) es = sol_edge[ed].size;
        aux = 2 * sol_edge[ii].len / (es + ss);
        ratio = aux − floor(aux);
        if(ratio < 0.5) k = (int) floor(aux);
        else k = (int) ceil(aux);
        if(k & 1) k−−;
        ratio = (es − ss) / (k − 1);

        aux = ss;
        t = ss;
        for(j=1 ; j < k ; j++){
          t = t + ratio;
          aux = aux + t;
        }
        aux = (sol_edge[ii].len - aux) / (REAL) k;
        tx = (sol_edge[ii].e_par -sol_edge[ii].s_par) / sol_edge[ii].len;
        rt = ratio * tx;
        dt = (ss + aux) * tx;
        t = sol_edge[ii].s_par + dt;
      }
    }
    sol_edge[ii].nel = k;
    for(i=1 ; i < k ; i++){
      ap_edgeparm2pt(sol_id, sol_edge[ii].id, t, node[ni].coord);
      node[ni].no = ni;
      edge[ei].node[1] = ni;
      edge[ei].no = ei;
      ei++;
      edge[ei].node[0] = ni;
      edge[ei].no = ei;
      edge[ei].edge_id = sol_edge[ii].id;
      ni++;
      dt = dt + rt;
      t = t + dt;
    }
    for(i=0 ; i < vi ; i++){
      if(dist(node[i].coord, sol_edge[ii].e_p) < tol){
        edge[ei].node[1] = i;
        edge[ei].no = ei;
        ei++;
        break;
      }
    }
  }
}
```

A.5 Function *initial_2D_front()*

```
void initial_2D_front()
{
  int i, ii, j, k, s, e, cs, ce;
  int **auxf;                   /* AUXILIARY ARRAY OF NODES IN THE FACE'S EDGES */
  int *auxe;    /* AUXILIARY ARRAY OF EDGES ALREADY ADDED TO THE PAVING FRONT */
  int ***con;                       /* CONNECTIVITY ARRAY OF THE FACE'S EDGES */
  REAL t, dt;
  auxe = (int *) calloc (sol_face[m].ne, sizeof (int));
  auxf = (int **) calloc (sol_face[m].ne, sizeof (int));
  for(i=0 ; i < sol_face[m].ne ; i++){
    auxf[i] = (int *) calloc ((sol_face[m].edge[i]->nel + 1), sizeof (int));
  }
  for(i=0 ; i < sol_face[m].ne ; i++){
    for(k=0 ; k < ei ; k++)
    if (edge[k].edge_id == sol_face[m].edge[i]->id) break;
  for(j=0 ; j < sol_face[m].edge[i]->nel ; j++)
    auxf[i][j] = edge[j+k].node[0];
  auxf[i][sol_face[m].edge[i]->nel] = edge[j+k-1].node[1];
}
con = (int ***) calloc (sol_face[m].ne, sizeof (int));
for(j=0 ; j < sol_face[m].ne ; j++){
  con[j] = (int **) calloc (2, sizeof (int));
  con[j][0] = (int *) calloc (2, sizeof (int));
  con[j][1] = (int *) calloc (2, sizeof (int));
}
for(j=0 ; j < sol_face[m].ne ; j++){
  s = 0;
  e = 0;
  i = 0;
  if(i == j) i++;
  while(((s == 0) || (e == 0)) && (i < sol_face[m].ne)){
    if(auxf[j][0] == auxf[j][sol_face[m].edge[j]->nel]){/*CHECK CLOSED EDGE*/
      con[j][0][0] = j;
      con[j][0][1] = 1;
      con[j][1][0] = j;
      con[j][1][1] = 0;
      break;
    }
    if(auxf[j][0] == auxf[i][0]){
      con[j][0][0] = i;
      con[j][0][1] = 0;
      s = 1;
    }
    if(auxf[j][0] == auxf[i][sol_face[m].edge[i]->nel]){
      con[j][0][0] = i;
      con[j][0][1] = 1;
      s = 1;
    }
    if(auxf[j][sol_face[m].edge[j]->nel] == auxf[i][0]){
      con[j][1][0] = i;
      con[j][1][1] = 0;
      e = 1;
    }
    if(auxf[j][sol_face[m].edge[j]->nel] ==
       auxf[i][sol_face[m].edge[i]->nel]){
      con[j][1][0] = i;
      con[j][1][1] = 1;
      e = 1;
    }
    i++;
    if(i == j) i++;
  }
}
for(j=0 ; j < MAX_FRONT ; j++) f[m][j].f  = -1;
pi[m] = 0;
n_f[m] = 0;
i = 0;
while(i < sol_face[m].ne){
    for(ii=0 ; ii < sol_face[m].ne ; ii++) {if(auxe[ii] == 0) break;}
    f[m][pi[m]].f = n_f[m];
    for(j=0 ; j<sol_face[m].edge[ii]->nel ; j++) f[m][j+pi[m]].n = auxf[ii][j];
    cs = f[m][pi[m]].n;
    pi[m] = pi[m] + sol_face[m].edge[ii]->nel - 1;
    auxe[i] = 1;
    i++;
    if(auxf[ii][0] == auxf[ii][sol_face[m].edge[ii]->nel]){
      f[m][pi[m]].f = n_f[m];
      n_f[m]++;
      pi[m]++;
    }
    else{
      e = 1;
      for( ; ; ){
        pi[m]++;
        if(con[ii][e][1] == 0){
          for(j=0 ; j < sol_face[m].edge[con[ii][e][0]]->nel ; j++)
            f[m][j+pi[m]].n = auxf[con[ii][e][0]][j];
          pi[m] = pi[m] + sol_face[m].edge[con[ii][e][0]]->nel - 1;
          ce = auxf[con[ii][e][0]][sol_face[m].edge[con[ii][e][0]]->nel];
          ii = con[ii][e][0];
          auxe[ii] = 1;
          e = 1;
        }
        else{
          k = sol_face[m].edge[con[ii][e][0]]->nel;
          for(j=0 ; j < sol_face[m].edge[con[ii][e][0]]->nel ; j++){
            f[m][j+pi[m]].n = auxf[con[ii][e][0]][k];
            k--;
          }
          pi[m] = pi[m] + sol_face[m].edge[con[ii][e][0]]->nel - 1;
          ce = auxf[con[ii][e][0]][0];
          ii = con[ii][e][0];
          auxe[ii] = 1;
          e = 0;
        }
        i++;
        if(ce == cs){
          f[m][pi[m]].f = n_f[m];
          n_f[m]++;
          pi[m]++;
          break;
        }
      }
    }
  }
  for(i=0 ; i < sol_face[m].ne ; i++){
    free(auxf[i]);
    free(con[i][0]);
    free(con[i][1]);
    free(con[i]);
  }
  free(auxf);
  free(con);
  free(auxe);
  pi[m]--;
  n_f[m]--;
}
```

A.6 Function *orientation(int nf)*

```
void orientation(int nf)
{
  int i;
  REAL dv, dt, d0, d1;
  ap_Param parm;
  POINT v, v0, v1, t, pt1;
  for(i=0 ; i <= pi[m] ; i++)
    if(f[m][i].f == nf) break;
  vector(i+1, i, v0);
  vector(i+1, i+2, v1);
  d0 = vector_norm(v0);
  d1 = vector_norm(v1);
  scalar_vector((d1 / d0), v0);
  sub_vector(v0, v1, v);
  ap_pt_norm2face(sol_id, sol_face[m].id, node[f[m][i+1].n].coord, TRUE, n[m]);
  vector_product(v, n[m], t);
  dt = vector_norm(t);
  scalar_vector((d1 / (40.0 * dt)), t);
  point_vector(i+1, t, pt1);
  point_projection(pt1);
  if(ap_facept2parm(sol_id, sol_face[m].id, pt1, &parm) == AP_NORMAL)
    turn[m][nf] = 1.0;
  else
    turn[m][nf] = -1.0;
}
```

A.7 Function *front_angle_distance(int nf)*

```
/* CALCULATE ANGLES AND DISTANCES OF PAVING FRONT */
void front_angle_distance(int nf)
{
  int i, j;
  REAL a0, a1, d0, d1, dt;
  POINT v, t, v0, v1;
  set_ki_kpi(nf);
  vector(ki, kpi, v0);
  vector(ki, ki+1, v1);
  d0 = vector_norm(v0);
  d1 = vector_norm(v1);
  scalar_vector((d1 / d0), v0);
  sub_vector(v0, v1, v);
  if(sol_face[m].type != 0)
    ap_pt_norm2face(sol_id, sol_face[m].id, node[f[m][ki].n].coord,TRUE, n[m]);
  vector_product(v, n[m], t);
  dt = vector_norm(t);
  scalar_vector(turn[m][nf], t);
  a0 = inner_product(t,v0) / (dt * d1);
  a1 = inner_product(t,v1) / (dt * d1);
  f[m][ki].a = acos(a0) + acos(a1);
  f[m][ki].d = d1;
  for(j=ki+1 ; j < kpi ; j++){
    vector(j, j-1, v0);
    vector(j, j+1, v1);
    d0 = d1;
    d1 = vector_norm(v1);
    scalar_vector((d1 / d0), v0);
    sub_vector(v0, v1, v);
    if(sol_face[m].type != 0)
      ap_pt_norm2face(sol_id, sol_face[m].id,node[f[m][j].n].coord,TRUE, n[m]);
    vector_product(v, n[m], t);
    dt = vector_norm(t);
    scalar_vector(turn[m][nf], t);
    a0 = inner_product(t,v0) / (dt * d1);
    a1 = inner_product(t,v1) / (dt * d1);
    f[m][j].a = acos(a0) + acos(a1);
    f[m][j].d = d1;
  }
  vector(kpi, kpi-1, v0);
  vector(kpi, ki, v1);
  d0 = d1;
  d1 = vector_norm(v1);
  scalar_vector((d1 / d0), v0);
  sub_vector(v0, v1, v);
  if(sol_face[m].type != 0)
    ap_pt_norm2face(sol_id, sol_face[m].id,node[f[m][kpi].n].coord,TRUE, n[m]);
  vector_product(v, n[m], t);
  dt = vector_norm(t);
  scalar_vector(turn[m][nf], t);
  a0 = inner_product(t,v0) / (dt * d1);
  a1 = inner_product(t,v1) / (dt * d1);
  f[m][kpi].a = acos(a0) + acos(a1);
  f[m][kpi].d = d1;
}

/* ANGLES AND DISTANCES OF A PAVING FRONT NODE */
void front_node_ang_dist(int nf, int i)
{
  int i0, i1;
  REAL a0, a1, d0, d1, dt;
  POINT t, v, v0, v1;
  i0 = decr(i);
  i1 = incr(i);
  vector(i, i0, v0);
  vector(i, i1, v1);
  d0 = vector_norm(v0);
  d1 = vector_norm(v1);
  scalar_vector((d1 / d0), v0);
  sub_vector(v0, v1, v);
  if(sol_face[m].type != 0)
    ap_pt_norm2face(sol_id, sol_face[m].id, node[f[m][i].n].coord, TRUE, n[m]);
  vector_product(v, n[m], t);
  dt = vector_norm(t);
  scalar_vector(turn[m][nf], t);
  a0 = inner_product(t,v0) / (dt * d1);
  a1 = inner_product(t,v1) / (dt * d1);
  f[m][i].a = acos(a0) + acos(a1);
  f[m][i].d = d1;
}
```

A.8 Function *close_corner(int nf)*

```
int close_corner(int nf)
{
  int in = 0, i, j, k0, k1, k2, k3, k4, w, k, kk;
  REAL d1, d2, d, dm;
  POINT v, v0, v1, P1;
  printf("   close_corner = ");
  set_ki_kpi(nf);
  for(i=ki ; i <= kpi ; i++){
    k0 = decr(i);
    k1 = incr(i);
    k2 = incr(k1);
    vector(i, k0, v0);
    vector(i, k1, v1);
    add_vector(v0, v1, v);
    scalar_vector(0.6, v);
    point_vector(i, v, P1);
    if(sol_face[m].type != 0) point_projection(P1);
    d1 = dist(node[f[m][k0].n].coord, P1);
    d2 = (f[m][k1].d + f[m][k2].d) / 2.0;
    d = dist(node[f[m][k0].n].coord, node[f[m][k2].n].coord);
    if((f[m][i].a <= a135) && (f[m][k1].a <= a135) && (d < 1.05*(d1+d2))){
      in = 1;
      create_face(f[m][k2].n, f[m][k0].n, f[m][i].n, f[m][k1].n);
      w = 0;
      if(i == kpi){
        kpi = kpi - 2;
        for(j=ki ; j <= kpi ; j++){
          f[m][j].n = f[m][j+1].n;
          f[m][j].a = f[m][j+1].a;
          f[m][j].d = f[m][j+1].d;
        }
        f[m][kpi].f = nf;
        f[m][kpi+1].f = nf;
        front_node_ang_dist(nf, ki);
        front_node_ang_dist(nf, kpi);
        w = 1;
      }
      if(i == kpi-1){
        kpi = kpi - 2;
        f[m][kpi].f = nf;
        f[m][kpi+1].f = nf;
        front_node_ang_dist(nf, ki);
        front_node_ang_dist(nf, kpi);
        w = 1;
      }
      if(w == 0){
        kpi = kpi - 2;
        for(j=i ; j <= kpi ; j++){
          f[m][j].n = f[m][j+2].n;
          f[m][j].a = f[m][j+2].a;
          f[m][j].d = f[m][j+2].d;
        }
        f[m][kpi].f = nf;
        f[m][kpi+1].f = nf;
        front_node_ang_dist(nf, i);
        front_node_ang_dist(nf, decr(i));
      }
      break;
    }
  }
  return in;
}
```

A.9 Function *seam(int nf)*

```
int seam(int nf)
{
  int i, j, jj, in = 0, k0, k1, e0, e1, w, anf;
  printf("   seam = ");
  set_ki_kpi(nf);
  for(i=ki ; i <= kpi ; i++){
    k0 = decr(i);
    k1 = incr(i);
    if(f[m][i].a <= a45){
      in = 1;
      node[f[m][k0].n].coord[X] = (node[f[m][k1].n].coord[X]
                                  + node[f[m][k0].n].coord[X]) / 2.0;
      node[f[m][k0].n].coord[Y] = (node[f[m][k1].n].coord[Y]
                                  + node[f[m][k0].n].coord[Y]) / 2.0;
      node[f[m][k0].n].coord[Z] = (node[f[m][k1].n].coord[Z]
                                  + node[f[m][k0].n].coord[Z]) / 2.0;
```

```
if(sol_face[m].type != 0) point_projection(node[f[m][k0].n].coord);
for(j=0 ; j < node[f[m][k1].n].nfi ; j++){
  node[f[m][k0].n].nface[node[f[m][k0].n].nfi]=node[f[m][k1].n].nface[j];
  node[f[m][k0].n].nfi++;
  for(jj=0 ; jj < 4 ; jj++){
    if(face[node[f[m][k1].n].nface[j]].fnode[jj] == f[m][k1].n){
      face[node[f[m][k1].n].nface[j]].fnode[jj] = f[m][k0].n;
      break;
    }
  }
}
for(j=0 ; j <= pi[m] ; j++){
  if((f[m][j].n == f[m][k1].n) && (j != k1) &&
     ((f[m][incr(j)].f == -1) || (f[m][decr(j)].f == -1))){
    jj = j;
    while(f[m][jj].f == -1) jj = decr(jj);
    anf = f[m][jj].f;
    f[m][j].n = f[m][k0].n;
    set_ki_kpi(anf);
    front_node_ang_dist(anf, decr(j));
    front_node_ang_dist(anf, j);
    front_node_ang_dist(anf, incr(j));
    set_ki_kpi(nf);
    continue;
  }
}
node[f[m][k1].n].no = -1;
node[f[m][k1].n].coord[X] = 1E+10;
node[f[m][k1].n].coord[Y] = 1E+10;
node[f[m][k1].n].coord[Z] = 1E+10;
w = 0;
if(i == kpi){
  kpi = kpi - 2;
  for(j=ki ; j <= kpi ; j++){
    f[m][j].n = f[m][j+1].n;
    f[m][j].a = f[m][j+1].a;
    f[m][j].d = f[m][j+1].d;
  }
  f[m][kpi].f = nf;
  f[m][kpi+1].f = nf;
  front_node_ang_dist(nf, ki);
  front_node_ang_dist(nf, kpi);
  front_node_ang_dist(nf, kpi-1);
  w = 1;
}
if(i == kpi-1){
  kpi = kpi - 2;
  f[m][kpi].f = nf;
  f[m][kpi+1].f = nf;
  front_node_ang_dist(nf, ki);
  front_node_ang_dist(nf, kpi);
  front_node_ang_dist(nf, kpi-1);
  w = 1;
}
```

```
      if(w == 0){
        kpi = kpi - 2;
        for(j=i ; j <= kpi ; j++){
          f[m][j].n = f[m][j+2].n;
          f[m][j].a = f[m][j+2].a;
          f[m][j].d = f[m][j+2].d;
        }
        f[m][kpi].f = nf;
        f[m][kpi+1].f = nf;
        front_node_ang_dist(nf, i);
        front_node_ang_dist(nf, decr(i));
        front_node_ang_dist(nf, decr(decr(i)));
      }
      break;
    }
  }
  return in;
}
```

A.10 Auxiliary functions to advance the 2D-front

```
/* CREATE FIRST FACE OF THE FRONT */
void create_first_face_of_front(int nf, int *ee, int *ss)
{
  int e, s;
  REAL d, d1, d0, dv;
  POINT v, v0, v1, P1;
  e = *ee;
  s = *ss;
  node[ni].no = ni;
  ff[afi] = ni;
  afi++;
  vector(e, decr(e), v0);
  vector(e, s, v1);
  sub_vector(v0, v1, v);
  if(sol_face[m].type != 0)
    ap_pt_norm2face(sol_id, sol_face[m].id, node[f[m][e].n].coord, TRUE, n[m]);
  vector_product(v, n[m], P1);
  d0 = vector_norm(v0);
  d1 = vector_norm(v1);
  d  = vector_norm(P1);
  if(f[m][decr(decr(decr(e)))].a <= a60) d = 8.0 * d / 7.0;
  if(f[m][decr(decr(e))].a <= a60) d = 8.0 * d / 5.0;
  if(f[m][decr(e)].a <= a60) d = d / sin(f[m][decr(e)].a/2.0);
  if(f[m][incr(s)].a <= a60) d = 8.0 * d / 5.0;
  if(f[m][incr(incr(s))].a <= a60) d = 8.0 * d / 7.0;
  scalar_vector((turn[m][nf]*(d0+d1)/(2.0*d)), P1);
  point_vector(e, P1, node[ni].coord);
  if(sol_face[m].type != 0) point_projection(node[ni].coord);
  ni++;
  node[ni].no = ni;
  ff[afi] = ni;
  afi++;
  vector(s, e, v0);
  vector(s, incr(s), v1);
  sub_vector(v0, v1, v);
  if(sol_face[m].type != 0)
    ap_pt_norm2face(sol_id, sol_face[m].id, node[f[m][s].n].coord, TRUE, n[m]);
  vector_product(v, n[m], P1);
  d0 = vector_norm(v0);
  d1 = vector_norm(v1);
  d  = vector_norm(P1);
  if(f[m][decr(decr(e))].a <= a60) d = 8.0 * d / 7.0;
  if(f[m][decr(e)].a <= a60) d = 8.0 * d / 5.0;
  if(f[m][incr(s)].a <= a60) d = d / sin(f[m][incr(s)].a/2.0);
  if(f[m][incr(incr(s))].a <= a60) d = 8.0 * d / 5.0;
  if(f[m][incr(incr(incr(s)))].a <= a60) d = 8.0 * d / 7.0;
  scalar_vector((turn[m][nf]*(d0+d1)/(2.0*d)), P1);
  point_vector(s, P1, node[ni].coord);
  if(sol_face[m].type != 0) point_projection(node[ni].coord);
  create_face(ni, ni-1, f[m][e].n, f[m][s].n);
  ni++;
  ee = &e;
  ss = &s;
}

/* DIVIDE FRONT INTO TWO */
void divide_front_into_two(int nf)
{
  int i, im, jm, k, w, k0, nk;
  REAL a;
  a = 0.0;
  for(i=ki ; i <=kpi ; i++) if(f[m][i].a > a) {a = f[m][i].a; im = i;}
  k0 = kpi - ki + 1;
  jm = incr(im);
  for(i=1 ; i < k0/2 ; i++) jm = incr(jm);
  nk = k0 / 2 - 3;
  for(i=0 ; i < nk ; i++){
    node[ni].no = ni;
    node[ni].coord[X] = ((nk - i) * node[f[m][im].n].coord[X] +
                        (i + 1) * node[f[m][jm].n].coord[X]) / (nk + 1);
    node[ni].coord[Y] = ((nk - i) * node[f[m][im].n].coord[Y] +
                        (i + 1) * node[f[m][jm].n].coord[Y]) / (nk + 1);
    node[ni].coord[Z] = ((nk - i) * node[f[m][im].n].coord[Z] +
                        (i + 1) * node[f[m][jm].n].coord[Z]) / (nk + 1);
    if(sol_face[m].type != 0) point_projection(node[ni].coord);
    ni++;
  }
  k = 0;
  i = im;
  while(i != jm){
    k++;
    f[m][pi[m]+k].n = f[m][i].n;
    f[m][pi[m]+k].a = f[m][i].a;
```

```
      f[m][pi[m]+k].f = -1;
      f[m][pi[m]+k].d = f[m][i].d;
      i = incr(i);
    }
    k++;
    f[m][pi[m]+k].n = f[m][jm].n;
    f[m][pi[m]+k].a = f[m][jm].a;
    f[m][pi[m]+k].f = -1;
    f[m][pi[m]+k].d = f[m][jm].d;
    n_f[m]++;
    pff[n_f[m]] = pff[n_f[m]-1];
    set_origin(n_f[m], nf);
    f[m][pi[m]+1].f = n_f[m];
    pi[m] = pi[m] + k + nk;
    for(i=0 ; i < nk ; i++){
      f[m][pi[m]-i].n = ni-(nk-i);
      f[m][pi[m]-i].f = -1;
    }
    f[m][pi[m]].f = n_f[m];
    turn[m][n_f[m]] = turn[m][nf];
    for(i=0 ; i < nk ; i++) af[i].n = ni-(nk-i);
    w = nk;
    i = jm;
    while(i != im){
      af[w].n = f[m][i].n;
      af[w].a = f[m][i].a;
      af[w].d = f[m][i].d;
      w++;
      i = incr(i);
    }
    af[w].n = f[m][im].n;
    af[w].a = f[m][im].a;
    af[w].d = f[m][im].d;
    w++;
    for(i=0 ; i < w ; i++){
      f[m][ki+i].n = af[i].n;
      f[m][ki+i].a = af[i].a;
      f[m][ki+i].d = af[i].d;
    }
    kpi = ki + w - 1;
    f[m][kpi].f = nf;
    k = k0 - w;
    for(i=kpi+1 ; i <= kpi+k ; i++) f[m][i].f = nf;
    front_node_ang_dist(nf, kpi);
    for(i=0 ; i <= nk ; i++) front_node_ang_dist(nf, ki+i);
    set_ki_kpi(n_f[m]);
    for(i=0 ; i <= nk ; i++) front_node_ang_dist(nf, kpi-i);
    front_node_ang_dist(n_f[m],ki);
}

/* CREATE FACE IN THE CORNER */
void face_in_corner(int nff, int i0, int i1, int i2)
{
    int i, ii, in=0;
```

```
REAL d0, d1, d, dv;
POINT v0, v1, v, p;
node[ni].no == ni;
ff[afi] = ni;
afi++;
vector(i1, i0, v0);
vector(i1, i2, v1);
sub_vector(v0, v1, v);
if(sol_face[m].type != 0)
  ap_pt_norm2face(sol_id, sol_face[m].id, node[f[m][i1].n].coord,TRUE, n[m]);
vector_product(v, n[m], p);
d0 = vector_norm(v0);
d1 = vector_norm(v1);
d  = vector_norm(p);
if(f[m][decr(decr(i0))].a <= a60) d = 8.0 * d / 7.0;
if(f[m][decr(i0)].a <= a60) d = 8.0 * d / 5.0;
if(f[m][i2].a <= a60) {d = d / sin(f[m][i2].a/2.0); in = 1;}
if(f[m][incr(i2)].a <= a60) d = 8.0 * d / 5.0;
if(f[m][incr(incr(i2))].a <= a60) d = 8.0 * d / 7.0;
if(in == 0){
  scalar_vector((turn[m][nff] * (d0+d1)/(2.0*d)), p);
  point_vector(i1, p, node[ni].coord);
  if(sol_face[m].type != 0) point_projection(node[ni].coord);
}
if(in == 1){
  d = (f[m][i1].d + f[m][i2].d) / (2.0 * cos(f[m][i2].a/2.0));
  ii = incr(i2);
  vector(i2, i1, v0);
  vector(i2, ii, v1);
  add_vector(v0, v1, v);
  dv = vector_norm(v);
  scalar_vector(d/dv, v);
  point_vector(i2, v, node[ni].coord);
  if(sol_face[m].type != 0) point_projection(node[ni].coord);
}
d = dist(node[ni-1].coord, node[ni].coord);
i = decr(i0);
if((f[m][i].a <= a135) && (d < f[m][i0].d/2.0)){
  ii = decr(i);
  vector(i, ii, v0);
  vector(i, i0, v1);
  add_vector(v0, v1, v);
  scalar_vector(0.6, v);
  point_vector(i, v, node[ni-1].coord);
  if(sol_face[m].type != 0) point_projection(node[ni-1].coord);
  if(repr == 1){
    ads_command(RTSTR, "erase", RTSTR, "l", RTSTR, "", 0);
    ads_command(RTSTR, "erase", RTSTR, "l", RTSTR, "", 0);
    ads_command(RTSTR, "3dface", RT3DPOINT, node[face[fi-2].fnode[0]].coord,
                RT3DPOINT, node[face[fi-2].fnode[1]].coord,
                RT3DPOINT, node[face[fi-2].fnode[2]].coord,
                RT3DPOINT, node[face[fi-2].fnode[3]].coord,
                RTSTR, "", 0);
    ads_command(RTSTR, "3dface", RT3DPOINT, node[face[fi-1].fnode[0]].coord,
                RT3DPOINT, node[face[fi-1].fnode[1]].coord,
                RT3DPOINT, node[face[fi-1].fnode[2]].coord,
                RT3DPOINT, node[face[fi-1].fnode[3]].coord,
                RTSTR, "", 0);
    }
  }
  create_face(ni, ni-1, f[m][i0].n, f[m][i1].n);
  ni++;
}

/* CREATE TWO FACES IN THE WEDGE */
void two_faces_in_wedge(int i0, int i1, int i2)
{
  REAL d, dv;
  POINT v0, v1, v;
  node[ni].no == ni;
  ff[afi] = ni;
  afi++;
  dv = (f[m][i0].d + f[m][i1].d) / 2.0;
  vector(i2, i1, v0);
  d = vector_norm(v0);
  scalar_vector(dv/d, v0);
  point_vector(i1, v0, node[ni].coord);
  if(sol_face[m].type != 0) point_projection(node[ni].coord);
  create_face(ni, ni-1, f[m][i0].n, f[m][i1].n);
  ni++;
  node[ni].no = ni;
  ff[afi] = ni;
  afi++;
  vector(i0, i1, v1);
  d = vector_norm(v1);
  scalar_vector(dv/d, v1);
  add_vector(v0, v1, v);
  d = vector_norm(v);
  scalar_vector(dv/d, v);
  point_vector(i1, v, node[ni].coord);
  if(sol_face[m].type != 0) point_projection(node[ni].coord);
  ni++;
  node[ni].no = ni;
  ff[afi] = ni;
  afi++;
  point_vector(i1, v1, node[ni].coord);
  if(sol_face[m].type != 0) point_projection(node[ni].coord);
  create_face(ni, ni-1, ni-2, f[m][i1].n);
  ni++;
}

/* CREATE THREE FACES IN THE WEDGE */
void three_faces_in_wedge(int i0, int i1, int i2)
{
  REAL d0, d1, d, dv;
  POINT v0, v1, v, P1, P2, P3, P4, P5, vP1, vP5;
  vector(i1, i0, v0);
  vector(i1, i2, v1);
```

```
add_vector(v0, v1, v);
d0 = f[m][i0].d;
d1 = f[m][i1].d;
d = (d0 + d1) / 2.0;
dv  = vector_norm(v);
scalar_vector((-1.0*(d/dv)), v);
point_vector(i1, v, P3);
if(sol_face[m].type != 0) point_projection(P3);
add_vector(v0, v, vP1);
dv = vector_norm(vP1);
scalar_vector(d/dv, vP1);
point_vector(i1, vP1, P1);
if(sol_face[m].type != 0) point_projection(P1);
add_vector(vP1, v, vP5);
dv = vector_norm(vP5);
scalar_vector(d/dv, vP5);
point_vector(i1, vP5, P2);
if(sol_face[m].type != 0) point_projection(P2);
add_vector(v1, v, vP5);
dv = vector_norm(vP5);
scalar_vector(d/dv, vP5);
point_vector(i1, vP5, P5);
if(sol_face[m].type != 0) point_projection(P5);
add_vector(vP5, v, vP1);
dv = vector_norm(vP1);
scalar_vector(d/dv, vP1);
point_vector(i1, vP1, P4);
if(sol_face[m].type != 0) point_projection(P4);
node[ni].no = ni;
ff[afi] = ni;
afi++;
node[ni].coord[X] = P1[X];
node[ni].coord[Y] = P1[Y];
node[ni].coord[Z] = P1[Z];
create_face(ni, ni-1, f[m][i0].n, f[m][i1].n);
ni++;
node[ni].no = ni;
ff[afi] = ni;
afi++;
node[ni].coord[X] = P2[X];
node[ni].coord[Y] = P2[Y];
node[ni].coord[Z] = P2[Z];
ni++;
node[ni].no = ni;
ff[afi] = ni;
afi++;
node[ni].coord[X] = P3[X];
node[ni].coord[Y] = P3[Y];
node[ni].coord[Z] = P3[Z];
create_face(ni, ni-1, ni-2, f[m][i1].n);
ni++;
node[ni].no = ni;
ff[afi] = ni;
afi++;
```

```
  node[ni].coord[X] = P4[X];
  node[ni].coord[Y] = P4[Y];
  node[ni].coord[Z] = P4[Z];
  ni++;
  node[ni].no = ni;
  ff[afi] = ni;
  afi++;
  node[ni].coord[X] = P5[X];
  node[ni].coord[Y] = P5[Y];
  node[ni].coord[Z] = P5[Z];
  create_face(ni, ni-1, ni-2, f[m][i1].n);
  ni++;
}

/* CORRECT CORNER FRONT */
void correct_corner_face(int i0, int i1, int i2)
{
  POINT v0, v1, v;
  vector(i1, i0, v0);
  vector(i1, i2, v1);
  add_vector(v0, v1, v);
  point_vector(i1, v, node[ni-1].coord);
  if(sol_face[m].type != 0) point_projection(node[ni-1].coord);
  if(repr == 1){
    ads_command(RTSTR, "erase", RTSTR, "l", RTSTR, "", 0);
    ads_command(RTSTR, "3dface", RT3DPOINT, node[face[fi-1].fnode[0]].coord,
                RT3DPOINT, node[face[fi-1].fnode[1]].coord,
                RT3DPOINT, node[face[fi-1].fnode[2]].coord,
                RT3DPOINT, node[face[fi-1].fnode[3]].coord,
                RTSTR, "", 0);
  }
}
```

A.11 Function *advance_front(int nf)*

```
int advance_front(int nf)
{
  int i, k, e0, i0, w, s, e, in=0;
  printf("   advance_front = ");
  afi=0;
  set_ki_kpi(nf);
  for(i=ki ; i <= kpi ; i++){
    if((a135 < f[m][i].a) && (f[m][i].a <= a240) &&
       (a135 < f[m][incr(i)].a) && (f[m][incr(i)].a <= a240)){
      s = incr(i);
      e = i;
      in = 1;
      break;
    }
  }
```

```
if(in == 1) create_first_face_of_front(nf, &e, &s);
else {divide_front_into_two(nf); return 0;}
i = incr(s);
e0 = decr(e);
w = 0;
for( ; ; ){
  if((i == e0) || (i == e)) break;
  if(f[m][i].a <= a135){
    if(w == 2) break;
    if(a90 < f[m][i].a) correct_corner_face(decr(i), i, incr(i));
    create_face( f[m][incr(i)].n, ni-1, f[m][decr(i)].n, f[m][i].n);
    i = incr(incr(i));
    w++;
    continue;
  }
  if((a135 < f[m][i].a) && (f[m][i].a <= a240)){
    face_in_corner(nf, decr(i), i, incr(i));
    i = incr(i);
    w = 0;
    continue;
  }
  if((a240 < f[m][i].a) && (f[m][i].a <= a300)){
    two_faces_in_wedge(decr(i), i, incr(i));
    i = incr(i);
    w = 0;
    continue;
  }
  if(f[m][i].a > a300){
    three_faces_in_wedge(decr(i), i, incr(i));
    i = incr(i);
    w = 0;
    continue;
  }
}
if(i == e){
  create_face(ff[0], ni-1, f[m][e0].n, f[m][i].n);
  node[ff[0]].nface[1] = node[ff[0]].nface[0];
  node[ff[0]].nface[0] = fi-1;
  node[f[m][e].n].nface[node[f[m][e].n].nfi-1] = node[ff[0]].nface[1];
  node[f[m][e].n].nface[node[f[m][e].n].nfi-2] = fi-1;
}
if((i == e0) && (w == 0)){
  if(f[m][i].a <= a135){
    i0 = decr(i);
    face[fi-1].fnode[0] = ff[0];
    if(a90 < f[m][i].a){
      correct_corner_face(i0, i, e);
      node[ff[0]].coord[X] = node[ni-1].coord[X];
      node[ff[0]].coord[Y] = node[ni-1].coord[Y];
      node[ff[0]].coord[Z] = node[ni-1].coord[Z];
      if(repr == 1) reprint_face(pff[nf]);
    }
    else{
      if(repr == 1){
        ads_command(RTSTR, "erase", RTSTR, "l", RTSTR, "", 0);
        ads_command(RTSTR,"3dface",RT3DPOINT,
                    node[face[fi-1].fnode[0]].coord,
                    RT3DPOINT, node[face[fi-1].fnode[1]].coord,
                    RT3DPOINT, node[face[fi-1].fnode[2]].coord,
                    RT3DPOINT, node[face[fi-1].fnode[3]].coord,
                    RTSTR, "", 0);
      }
    }
    ni--;
    node[ni].nfi = 0;
    afi--;
    create_face(ff[0], f[m][i0].n, f[m][e0].n, f[m][e].n);
    node[ff[0]].nfi++;
    node[ff[0]].nface[2] = node[ff[0]].nface[0];
    node[ff[0]].nface[0] = fi-2;
    node[ff[0]].nface[1] = fi-1;
  }
  if((a135 < f[m][i].a) && (f[m][i].a <= a240)){
    face_in_corner(nf, decr(i), i, e);
    create_face(ff[0], ni-1, f[m][e0].n, f[m][e].n);
    node[ff[0]].nface[1] = node[ff[0]].nface[0];
    node[ff[0]].nface[0] = fi-1;
    node[f[m][e].n].nface[node[f[m][e].n].nfi-1] = node[ff[0]].nface[1];
    node[f[m][e].n].nface[node[f[m][e].n].nfi-2] = fi-1;
  }
  if((a240 < f[m][i].a) && (f[m][i].a <= a300)){
    two_faces_in_wedge(decr(i), i, e);
    create_face(ff[0], ni-1, f[m][e0].n, f[m][e].n);
    node[ff[0]].nface[1] = node[ff[0]].nface[0];
    node[ff[0]].nface[0] = fi-1;
    node[f[m][e].n].nface[node[f[m][e].n].nfi-1] = node[ff[0]].nface[1];
    node[f[m][e].n].nface[node[f[m][e].n].nfi-2] = fi-1;
  }
  if(f[m][i].a > a300){
    three_faces_in_wedge(decr(i), i, e);
    create_face(ff[0], ni-1, f[m][e0].n, f[m][e].n);
    node[ff[0]].nface[1] = node[ff[0]].nface[0];
    node[ff[0]].nface[0] = fi-1;
    node[f[m][e].n].nface[node[f[m][e].n].nfi-1] = node[ff[0]].nface[1];
    node[f[m][e].n].nface[node[f[m][e].n].nfi-2] = fi-1;
  }
}
if((i == e0) && (w > 0)){
  i0 = decr(i);
  if(dist(node[ni-1].coord, node[ff[0]].coord) <
     (f[m][i0].d + f[m][decr(i0)].d) / 3.0){
    ni--;
    node[ni].nfi = 0;
    afi--;
    face[fi-1].fnode[1] = ff[0];
    face[fi-2].fnode[0] = ff[0];
    if(repr == 1){
      ads_command(RTSTR, "erase", RTSTR, "l", RTSTR, "", 0);
      ads_command(RTSTR, "erase", RTSTR, "l", RTSTR, "", 0);
      ads_command(RTSTR, "3dface", RT3DPOINT,
                    node[face[fi-2].fnode[0]].coord,
                    RT3DPOINT, node[face[fi-2].fnode[1]].coord,
                    RT3DPOINT, node[face[fi-2].fnode[2]].coord,
                    RT3DPOINT, node[face[fi-2].fnode[3]].coord,
                    RTSTR, "", 0);
      ads_command(RTSTR, "3dface", RT3DPOINT,
                    node[face[fi-1].fnode[0]].coord,
                    RT3DPOINT, node[face[fi-1].fnode[1]].coord,
                    RT3DPOINT, node[face[fi-1].fnode[2]].coord,
                    RT3DPOINT, node[face[fi-1].fnode[3]].coord,
                    RTSTR, "", 0);
    }
    create_face(ff[0], f[m][i0].n, f[m][e0].n, f[m][e].n);
    node[ff[0]].nfi = node[ff[0]].nfi + 2;
    node[ff[0]].nface[3] = node[ff[0]].nface[0];
    node[ff[0]].nface[0] = fi-3;
    node[ff[0]].nface[1] = fi-2;
    node[ff[0]].nface[2] = fi-1;
  }
  else{
    if(f[m][i].a < a180){
      create_face(ff[0], f[m][i0].n, f[m][e0].n, f[m][e].n);
      ff[afi] = f[m][i0].n;
      afi++;
      node[ff[0]].nface[1] = node[ff[0]].nface[0];
      node[ff[0]].nface[0] = fi-1;
      node[f[m][e].n].nface[node[f[m][e].n].nfi-1] = node[ff[0]].nface[1];
      node[f[m][e].n].nface[node[f[m][e].n].nfi-2] = fi-1;
    }
    if((a180 <= f[m][i].a) && (f[m][i].a <= a240)){
      face_in_corner(nf, decr(i), i, e);
      create_face(ff[0], ni-1, f[m][e0].n, f[m][e].n);
      node[ff[0]].nface[1] = node[ff[0]].nface[0];
      node[ff[0]].nface[0] = fi-1;
      node[f[m][e].n].nface[node[f[m][e].n].nfi-1] = node[ff[0]].nface[1];
      node[f[m][e].n].nface[node[f[m][e].n].nfi-2] = fi-1;
    }
    if((a240 < f[m][i].a) && (f[m][i].a <= a300)){
      two_faces_in_wedge(decr(i), i, e);
      create_face(ff[0], ni-1, f[m][e0].n, f[m][e].n);
      node[ff[0]].nface[1] = node[ff[0]].nface[0];
      node[ff[0]].nface[0] = fi-1;
      node[f[m][e].n].nface[node[f[m][e].n].nfi-1] = node[ff[0]].nface[1];
      node[f[m][e].n].nface[node[f[m][e].n].nfi-2] = fi-1;
    }
    if(f[m][i].a > a300){
      three_faces_in_wedge(decr(i), i, e);
      create_face(ff[0], ni-1, f[m][e0].n, f[m][e].n);
      node[ff[0]].nface[1] = node[ff[0]].nface[0];
      node[ff[0]].nface[0] = fi-1;
      node[f[m][e].n].nface[node[f[m][e].n].nfi-1] = node[ff[0]].nface[1];
      node[f[m][e].n].nface[node[f[m][e].n].nfi-2] = fi-1;
      }
    }
  }
  if((w == 2) && (i != e) && (i != e0)){
    i = decr(i);
    while(i != s){
      ff[afi] = f[m][i].n;
      afi++;
      i = incr(i);
    }
  }
  afi--;
  if(afi == (kpi - ki)) for(i=0 ; i <= afi ; i++) f[m][i+ki].n = ff[i];
  if(afi > (kpi - ki)){
    k = afi - (kpi - ki);
    for(i = pi[m] ; i > kpi ; i--){
      f[m][i+k].n = f[m][i].n;
      f[m][i+k].f = f[m][i].f;
      f[m][i+k].a = f[m][i].a;
      f[m][i+k].d = f[m][i].d;
    }
    for(i=kpi ; i < kpi+k ; i++) f[m][i].f = -1;
    f[m][kpi+k].f = nf;
    for(i=0 ; i <= afi ; i++) f[m][i+ki].n = ff[i];
    kpi = ki + afi;
    pi[m] = pi[m] + k;
  }
  if(afi < (kpi - ki)){
    for(i=0 ; i <= afi ; i++) f[m][i+ki].n = ff[i];
    for(i=ki+afi ; i <= kpi ; i++) f[m][i].f = nf;
    kpi = ki + afi;
  }
  front_angle_distance(nf);
  return 1;
}
```

A.12 Auxiliary functions to correct the element sizes

```
/* JOINT TWO FACES */
void joint_two_faces(int i0, int i1, int i2)
{
  int i, j, f0, f1, f2, f3, n3, i3;
  REAL d, dv;
  POINT v;
  f0 = node[f[m][i0].n].nface[0];
  f1 = node[f[m][i1].n].nface[0];
  f2 = node[f[m][i2].n].nface[0];
  f3 = node[f[m][i2].n].nface[1];
  n3 = face[f1].fnode[3];
  node[f[m][i0].n].coord[X] = (node[f[m][i2].n].coord[X] +
                               node[f[m][i0].n].coord[X]) / 2.0;
  node[f[m][i0].n].coord[Y] = (node[f[m][i2].n].coord[Y] +
                               node[f[m][i0].n].coord[Y]) / 2.0;
  node[f[m][i0].n].coord[Z] = (node[f[m][i2].n].coord[Z] +
                               node[f[m][i0].n].coord[Z]) / 2.0;
  if(sol_face[m].type != 0) point_projection(node[f[m][i0].n].coord);
  if(node[f[m][i3=incr(i2)].n].nfi == 1) face[f3].fnode[2] = f[m][i0].n;
  else face[f3].fnode[1] = f[m][i0].n;
  face[f2].no = -1;
  face[f2].fnode[0] = -face[f2].fnode[0];
  face[f2].fnode[1] = -face[f2].fnode[1];
  face[f2].fnode[2] = -face[f2].fnode[2];
  face[f2].fnode[3] = -face[f2].fnode[3];
  if(repr == 1) ads_command(RTSTR, "erase", RTSTR, "l", RTSTR, "", 0);
  node[f[m][i1].n].no = -1;
  node[f[m][i1].n].coord[X] = 1E+10;
  node[f[m][i1].n].coord[Y] = 1E+10;
  node[f[m][i1].n].coord[Z] = 1E+10;
  node[f[m][i2].n].no = -1;
  node[f[m][i2].n].coord[X] = 1E+10;
  node[f[m][i2].n].coord[Y] = 1E+10;
  node[f[m][i2].n].coord[Z] = 1E+10;
  face[f1].fnode[0] = f[m][i0].n;
  face[f1].fnode[1] = face[f0].fnode[3];
  face[f1].fnode[2] = n3;
  face[f1].fnode[3] = face[f3].fnode[2];
  if(node[f[m][i3].n].nfi == 1) face[f1].fnode[3] = face[f3].fnode[3];
  node[f[m][i0].n].nface[node[f[m][i0].n].nfi] = f3;
  node[f[m][i0].n].nfi++;
  node[n3].nfi--;
  if(node[f[m][i3].n].nfi == 1)
    node[face[f1].fnode[3]].nface[node[face[f1].fnode[3]].nfi-3] = f1;
  else
    node[face[f1].fnode[3]].nface[node[face[f1].fnode[3]].nfi-2] = f1;
}

/* INSERT A FACE */
void insert_face(int i0, int i1, int i2)
{
  int f0, f1;
  POINT v;
  f0 = node[f[m][i1].n].nface[0];
  f1 = node[f[m][i2].n].nface[0];
  node[ni].no = ni;
  node[ni].coord[X] = (node[f[m][i1].n].coord[X] +
                       node[f[m][i2].n].coord[X]) / 2.0;
  node[ni].coord[Y] = (node[f[m][i1].n].coord[Y] +
                       node[f[m][i2].n].coord[Y]) / 2.0;
  node[ni].coord[Z] = (node[f[m][i1].n].coord[Z] +
                       node[f[m][i2].n].coord[Z]) / 2.0;
  if(sol_face[m].type != 0) point_projection(node[ni].coord);
  ni++;
  vector_no(face[f0].fnode[3], f[m][i1].n, v);
  scalar_vector(0.4, v);
  node[ni].no = ni;
  node[ni].coord[X] = node[f[m][i1].n].coord[X] + v[X];
  node[ni].coord[Y] = node[f[m][i1].n].coord[Y] + v[Y];
  node[ni].coord[Z] = node[f[m][i1].n].coord[Z] + v[Z];
  if(sol_face[m].type != 0) point_projection(node[ni].coord);
  ni++;
  node[f[m][i1].n].coord[X] = (node[f[m][i1].n].coord[X] +
                               node[f[m][i0].n].coord[X]) / 2.0;
  node[f[m][i1].n].coord[Y] = (node[f[m][i1].n].coord[Y] +
                               node[f[m][i0].n].coord[Y]) / 2.0;
  node[f[m][i1].n].coord[Z] = (node[f[m][i1].n].coord[Z] +
                               node[f[m][i0].n].coord[Z]) / 2.0;
  if(sol_face[m].type != 0) point_projection(node[f[m][i1].n].coord);
  face[fi].no = fi;
  face[fi].fnode[0] = ni-2;
  face[fi].fnode[1] = ni-1;
  face[fi].fnode[2] = f[m][i1].n;
  face[fi].fnode[3] = face[f0].fnode[3];
  face[fi].fm = m;
  fi++;
  node[ni-1].nface[node[ni-1].nfi] = fi-1;
  node[ni-1].nfi++;
  node[ni-2].nface[node[ni-2].nfi] = fi-1;
  node[ni-2].nfi++;
  node[ni-2].nface[node[ni-2].nfi] = f1;
  node[ni-2].nfi++;
  node[f[m][i1].n].nface[node[f[m][i1].n].nfi-1] = fi-1;
  node[face[f0].fnode[3]].nface[node[face[f0].fnode[3]].nfi-1] = fi-1;
  node[face[f0].fnode[3]].nface[node[face[f0].fnode[3]].nfi] = f1;
  node[face[f0].fnode[3]].nfi++;
  face[f1].fnode[1] = ni-2;
  if(repr == 1){
    ads_command(RTSTR, "3dface", RT3DPOINT, node[face[fi-1].fnode[0]].coord,
                RT3DPOINT, node[face[fi-1].fnode[1]].coord,
                RT3DPOINT, node[face[fi-1].fnode[2]].coord,
                RT3DPOINT, node[face[fi-1].fnode[3]].coord,
                RTSTR, "", 0);
  }
}

/* INTERSECT TEST FOR correct_front_size */
int intersect_for_correct(int i)
{
  int j, i0, i1;
  REAL d;
  i0 = decr(i);
  i1 = incr(incr(i));
  if(f[m][i0].d < f[m][i].d) d = f[m][i].d;
  else d = f[m][i0].d;
  j = i1;
  while(j != i0){
    if(dist(node[f[m][i].n].coord, node[f[m][j].n].coord) <= d) return 1;
    j = incr(j);
  }
  return 0;
}
```

A.13 Function *correct_front_size(int nf)*

```
int correct_front_size(int nf)
{
  int i=ki, j, in=0, e, b, i1, i2, w;
  REAL tmin, tmax;
  printf("   correct_front_size = ");
  set_ki_kpi(nf);
  tmin = 0.7 * msize[m];
  tmax = 1.3 * msize[m];
  while(i <= kpi){
    e = 0;
    i1 = incr(i);
    i2 = incr(i1);
    b = 0;
    for(j=0 ; j < n_i_n ; j++)
      if((f[m][i].n == no_intersect_node[j]) ||
         (f[m][i1].n == no_intersect_node[j]) ||
         (f[m][i2].n == no_intersect_node[j])) b = 1;
    b = b + intersect_for_correct(i);
    b = b + intersect_for_correct(i1);
    b = b + intersect_for_correct(i2);
    if((f[m][i].d < tmin) && (f[m][i1].d < tmin) &&
       (node[f[m][i].n].nfi == 2) && (node[f[m][i1].n].nfi == 2) &&
       (node[f[m][i2].n].nfi == 2) && (b == 0)){
      in = 1;
      joint_two_faces(i, i1, i2);
      w = 0;
      if(i < kpi-1){
        kpi = kpi - 2;
        for(j=i+1 ; j <= kpi ; j++){
          f[m][j].n = f[m][j+2].n;
          f[m][j].a = f[m][j+2].a;
          f[m][j].d = f[m][j+2].d;
        }
        f[m][kpi].f = nf;
        f[m][kpi+1].f = nf;
        front_node_ang_dist(nf, decr(i));
        front_node_ang_dist(nf, i);
        front_node_ang_dist(nf, incr(i));
        w = 1;
      }
      if((i == kpi-1) && (w == 0)){
        kpi = kpi - 2;
        for(j=ki ; j <= kpi ; j++){
```

```
      f[m][j].n = f[m][j+1].n;
      f[m][j].a = f[m][j+1].a;
      f[m][j].d = f[m][j+1].d;
    }
    f[m][kpi].f = nf;
    f[m][kpi+1].f = nf;
    front_node_ang_dist(nf, kpi-1);
    front_node_ang_dist(nf, kpi);
    front_node_ang_dist(nf, ki);
    w = 1;
  }
  if((i == kpi) && (w == 0)){
    kpi = kpi - 2;
    for(j=ki ; j <= kpi ; j++){
      f[m][j].n = f[m][j+2].n;
      f[m][j].a = f[m][j+2].a;
      f[m][j].d = f[m][j+2].d;
    }
    f[m][kpi].f = nf;
    f[m][kpi+1].f = nf;
    front_node_ang_dist(nf, kpi-1);
    front_node_ang_dist(nf, kpi);
    front_node_ang_dist(nf, ki);
  }
  i = i + 3;
  if(i > kpi) break;
  e = 1;
}
i1 = incr(i);
i2 = incr(i1);
b = 0;
for(j=0 ; j < n_i_n ; j++)
  if((f[m][i].n == no_intersect_node[j]) ||
     (f[m][i1].n == no_intersect_node[j]) ||
     (f[m][i2].n == no_intersect_node[j])) b = 1;
b = b + intersect_for_correct(i1);
if((f[m][i].d > tmax) && (f[m][i1].d > tmax) &&
   (node[f[m][i1].n].nfi == 2) && (b == 0)){
  in = 1;
  insert_face(i, i1, i2);
  for(j=pi[m] ; j > kpi ; j--){
    f[m][j+2].n = f[m][j].n;
    f[m][j+2].a = f[m][j].a;
    f[m][j+2].d = f[m][j].d;
    f[m][j+2].f = f[m][j].f;
  }
  pi[m] = pi[m] + 2;
  w = 0;
  if(i < kpi-1){
    kpi = kpi + 2;
    for(j=kpi ; j >= i+4 ; j--){
      f[m][j].n = f[m][j-2].n;
      f[m][j].a = f[m][j-2].a;
      f[m][j].d = f[m][j-2].d;
```

```
          f[m][j].f = f[m][j-2].f;
        }
        f[m][kpi-1].f = -1;
        f[m][kpi-2].f = -1;
        f[m][kpi].f = nf;
        f[m][i+2].n = ni-1;
        f[m][i+3].n = ni-2;

        front_node_ang_dist(nf, i);
        front_node_ang_dist(nf, i+1);
        front_node_ang_dist(nf, i+2);
        front_node_ang_dist(nf, i+3);
        w = 1;
      }
      if((i == kpi-1) && (w == 0)){
        kpi = kpi + 2;
        f[m][kpi-2].f = -1;
        f[m][kpi-1].f = -1;
        f[m][kpi].f = nf;
        f[m][kpi-1].n = ni-1;
        f[m][kpi].n = ni-2;
        front_node_ang_dist(nf, kpi-3);
        front_node_ang_dist(nf, kpi-2);
        front_node_ang_dist(nf, kpi-1);
        front_node_ang_dist(nf, kpi);
        w = 1;
      }
      if((i == kpi) && (w == 0)){
        kpi = kpi + 2;
        for(j=kpi ; j >= ki+3 ; j--){
          f[m][j].n = f[m][j-2].n;
          f[m][j].a = f[m][j-2].a;
          f[m][j].d = f[m][j-2].d;
          f[m][j].f = f[m][j-2].f;
        }
        f[m][kpi-1].f = -1;
        f[m][kpi-2].f = -1;
        f[m][kpi].f = nf;
        f[m][ki+1].n = ni-1;
        f[m][ki+2].n = ni-2;
        front_node_ang_dist(nf, kpi);
        front_node_ang_dist(nf, ki);
        front_node_ang_dist(nf, ki+1);
        front_node_ang_dist(nf, ki+2);
      }
      i = i + 5;
      e = 1;
    }
    if(e == 0) i++;
  }
  if(in == 1) if(repr == 1) reprint_face(pfc);
  return in;
}
```

A.14 Auxiliary functions to predict intersection

```
/* INTERSECT WITH NEW NODE */
void intersect_with_new_node(int nf, int im, int jm)
{
  int j, k, k2;
  node[ni].no = ni;
  node[ni].coord[X] = (node[f[m][im].n].coord[X] +
                       node[f[m][jm].n].coord[X]) / 2.0;
  node[ni].coord[Y] = (node[f[m][im].n].coord[Y] +
                       node[f[m][jm].n].coord[Y]) / 2.0;
  node[ni].coord[Z] = (node[f[m][im].n].coord[Z] +
                       node[f[m][jm].n].coord[Z]) / 2.0;
  if(sol_face[m].type != 0) point_projection(node[ni].coord);
  ni++;
  k = 0;
  for(j=im ; j <= jm ; j++){
    k++;
    f[m][pi[m]+k].n = f[m][j].n;
    f[m][pi[m]+k].a = f[m][j].a;
    f[m][pi[m]+k].f = -1;
    f[m][pi[m]+k].d = f[m][j].d;
  }
  n_f[m]++;
  pff[n_f[m]] = pff[n_f[m]-1];
  set_origin(n_f[m], nf);
  f[m][pi[m]+1].f = n_f[m];
  pi[m] = pi[m] + k + 1;
  f[m][pi[m]].n = ni-1;
  f[m][pi[m]].f = n_f[m];
  turn[m][n_f[m]] = turn[m][nf];
  f[m][im+1].n = ni-1;
  k2 = 0;
  for(j=jm ; j <= kpi ; j++){
    k2++;
    f[m][im+1+k2].n = f[m][j].n;
    f[m][im+1+k2].a = f[m][j].a;
    f[m][im+1+k2].d = f[m][j].d;
  }
  kpi = im + 1 + k2;
  f[m][kpi].f = nf;
  k = k - 3;
  for(j=kpi+1 ; j <= pi[m]-k ; j++){
    f[m][j].n = f[m][j+k].n;
    f[m][j].a = f[m][j+k].a;
    f[m][j].f = f[m][j+k].f;
    f[m][j].d = f[m][j+k].d;
  }
  pi[m] = pi[m] - k;
  front_node_ang_dist(nf, im);
  front_node_ang_dist(nf, im+1);
  front_node_ang_dist(nf, im+2);
  set_ki_kpi(n_f[m]);
  front_node_ang_dist(n_f[m],kpi-1);
  front_node_ang_dist(n_f[m],kpi);
  front_node_ang_dist(n_f[m],ki);
}

/* INTERSECT MOVE NODES */
void intersect_move_nodes(REAL d, int nf, int im, int jm)
{
  int j, k, k2;
  REAL dn, dv;
  POINT v;
  dn = (d - msize[m]) / 2.0;
  vector(im, jm, v);
  dv = vector_norm(v);
  scalar_vector(dn/dv, v);
  node[f[m][im].n].coord[X] = node[f[m][im].n].coord[X] + v[X];
  node[f[m][im].n].coord[Y] = node[f[m][im].n].coord[Y] + v[Y];
  node[f[m][im].n].coord[Z] = node[f[m][im].n].coord[Z] + v[Z];
  if(sol_face[m].type != 0) point_projection(node[f[m][im].n].coord);
  node[f[m][jm].n].coord[X] = node[f[m][jm].n].coord[X] - v[X];
  node[f[m][jm].n].coord[Y] = node[f[m][jm].n].coord[Y] - v[Y];
  node[f[m][jm].n].coord[Z] = node[f[m][jm].n].coord[Z] - v[Z];
  if(sol_face[m].type != 0) point_projection(node[f[m][jm].n].coord);
  k = 0;
  for(j=im ; j <= jm ; j++){
    k++;
    f[m][pi[m]+k].n = f[m][j].n;
    f[m][pi[m]+k].a = f[m][j].a;
    f[m][pi[m]+k].f = -1;
    f[m][pi[m]+k].d = f[m][j].d;
  }
  n_f[m]++;
  pff[n_f[m]] = pff[n_f[m]-1];
  set_origin(n_f[m], nf);
  f[m][pi[m]+1].f = n_f[m];
  f[m][pi[m]+k].f = n_f[m];
  pi[m] = pi[m] + k;
  turn[m][n_f[m]] = turn[m][nf];
  k2 = 0;
  for(j=jm ; j <= kpi ; j++){
    k2++;
    f[m][im+k2].n = f[m][j].n;
    f[m][im+k2].a = f[m][j].a;
    f[m][im+k2].d = f[m][j].d;
  }
  kpi = im + k2;
  f[m][kpi].f = nf;
  k = k - 2;
  for(j=kpi+1 ; j <= pi[m]-k ; j++){
    f[m][j].n = f[m][j+k].n;
    f[m][j].a = f[m][j+k].a;
    f[m][j].f = f[m][j+k].f;
    f[m][j].d = f[m][j+k].d;
  }
  pi[m] = pi[m] - k;
  front_node_ang_dist(nf, decr(im));
  front_node_ang_dist(nf, im);
  front_node_ang_dist(nf, im+1);
  front_node_ang_dist(nf, im+2);
  set_ki_kpi(n_f[m]);
  front_node_ang_dist(n_f[m],kpi-1);
  front_node_ang_dist(n_f[m],kpi);
  front_node_ang_dist(n_f[m],ki);
  front_node_ang_dist(n_f[m],ki+1);
}

/* INTERSECT SEAM NODES */
void intersect_seam_nodes(int nf, int im, int jm)
{
  int i, j, k, k2, anf;
  node[f[m][im].n].coord[X] = (node[f[m][im].n].coord[X] +
                               node[f[m][jm].n].coord[X]) / 2.0;
  node[f[m][im].n].coord[Y] = (node[f[m][im].n].coord[Y] +
                               node[f[m][jm].n].coord[Y]) / 2.0;
  node[f[m][im].n].coord[Z] = (node[f[m][im].n].coord[Z] +
                               node[f[m][jm].n].coord[Z]) / 2.0;
  if(sol_face[m].type != 0) point_projection(node[f[m][im].n].coord);
  for(j=0 ; j < node[f[m][jm].n].nfi ; j++){
    node[f[m][im].n].nface[node[f[m][im].n].nfi] =
      node[f[m][jm].n].nface[j];
    node[f[m][im].n].nfi++;
    for(k=0 ; k < 4 ; k++){
      if(face[node[f[m][jm].n].nface[j]].fnode[k] == f[m][jm].n){
        face[node[f[m][jm].n].nface[j]].fnode[k] = f[m][im].n;
        break;
      }
    }
  }
  for(i=0 ; i <= pi[m] ; i++){
    if((f[m][i].n == f[m][jm].n) && (i != jm) &&
       ((f[m][i+1].f == -1) || (f[m][i-1].f == -1))){
      j = i;
      while(f[m][j].f == -1) j = decr(j);
      anf = f[m][j].f;
      f[m][i].n = f[m][im].n;
          set_ki_kpi(anf);
      front_node_ang_dist(anf, decr(i));
      front_node_ang_dist(anf, i);
      front_node_ang_dist(anf, incr(i));
      set_ki_kpi(nf);
      continue;
    }
  }
  node[f[m][jm].n].no = -1;
  node[f[m][jm].n].coord[X] = 1E+10;
  node[f[m][jm].n].coord[Y] = 1E+10;
  node[f[m][jm].n].coord[Z] = 1E+10;
  k = 0;
  for(j=im ; j < jm ; j++){
    k++;
    f[m][pi[m]+k].n = f[m][j].n;
    f[m][pi[m]+k].a = f[m][j].a;
    f[m][pi[m]+k].f = -1;
    f[m][pi[m]+k].d = f[m][j].d;
  }
  n_f[m]++;
  pff[n_f[m]] = pff[n_f[m]-1];
  set_origin(n_f[m], nf);
  f[m][pi[m]+1].f = n_f[m];
  f[m][pi[m]+k].f = n_f[m];
  pi[m] = pi[m] + k;
  turn[m][n_f[m]] = turn[m][nf];
  k2 = 0;
  for(j=jm+1 ; j <= kpi ; j++){
    k2++;
    f[m][im+k2].n = f[m][j].n;
    f[m][im+k2].a = f[m][j].a;
    f[m][im+k2].d = f[m][j].d;
  }
  kpi = im + k2;
  f[m][kpi].f = nf;
  for(j=kpi+1 ; j <= pi[m]-k ; j++){
    f[m][j].n = f[m][j+k].n;
    f[m][j].a = f[m][j+k].a;
    f[m][j].f = f[m][j+k].f;
    f[m][j].d = f[m][j+k].d;
  }
  pi[m] = pi[m] - k;
  front_node_ang_dist(nf, decr(im));
  front_node_ang_dist(nf, im);
  front_node_ang_dist(nf, im+1);
  set_ki_kpi(n_f[m]);
  front_node_ang_dist(n_f[m],kpi);
  front_node_ang_dist(n_f[m],ki);
  front_node_ang_dist(n_f[m],ki+1);
}

/* SET no_intersect_node */
void set_no_intersect_node(int i)
{
  no_intersect_node[n_i_n] = i;
  n_i_n++;
 }

/* CHECK no_intersect_node */
int check_no_intersect_node(int nf, int i, int j)
{
  int k;
  REAL d, d0, d1, a;
  POINT v, v0, v1, t;
  for(k=0 ; k < n_i_n ; k++)
```

```
      if((f[m][i].n == no_intersect_node[k]) ||
         (f[m][j].n == no_intersect_node[k])) return 1;
    if(nf != -1){
      vector(i, decr(i), v0);
      vector(i, incr(i), v1);
      d0 = vector_norm(v0);
      d1 = vector_norm(v1);
      scalar_vector((d1 / d0), v0);
      sub_vector(v0, v1, v);
      if(sol_face[m].type != 0)
        ap_pt_norm2face(sol_id, sol_face[m].id,node[f[m][j].n].coord,TRUE, n[m]);
      vector_product(v, n[m], t);
      scalar_vector(turn[m][nf], t);
      vector(i, j, v);
      a = inner_product(t,v);
      if(a < 0) return 1;
      vector(j, decr(j), v0);
      vector(j, incr(j), v1);
      d0 = vector_norm(v0);
      d1 = vector_norm(v1);
      scalar_vector((d1 / d0), v0);
      sub_vector(v0, v1, v);
      if(sol_face[m].type != 0)
        ap_pt_norm2face(sol_id, sol_face[m].id,node[f[m][j].n].coord,TRUE, n[m]);
      vector_product(v, n[m], t);
      scalar_vector(turn[m][nf], t);
      vector(j, i, v);
      a = inner_product(t,v);
      if(a < 0) return 1;
    }
    return 0;
}

/* INTERSECT TWO FRONTS WITH TWO NEW NODES */
void intersect_two_fronts_with_two_new_nodes(int ao, int nf,
                                          int im, int is, int jm, int js)
{
    int i, k, w;
    set_no_intersect_node(f[m][im].n);
    set_no_intersect_node(f[m][is].n);
    set_no_intersect_node(f[m][jm].n);
    set_no_intersect_node(f[m][js].n);
    node[ni].no = ni;
    node[ni].coord[X] = (node[f[m][im].n].coord[X] +
                         node[f[m][jm].n].coord[X]) / 2.0;
    node[ni].coord[Y] = (node[f[m][im].n].coord[Y] +
                         node[f[m][jm].n].coord[Y]) / 2.0;
    node[ni].coord[Z] = (node[f[m][im].n].coord[Z] +
                         node[f[m][jm].n].coord[Z]) / 2.0;
    if(sol_face[m].type != 0) point_projection(node[ni].coord);
    set_no_intersect_node(ni);
    ni++;
    node[ni].no = ni;
    node[ni].coord[X] = (node[f[m][is].n].coord[X] +
                        node[f[m][js].n].coord[X]) / 2.0;
node[ni].coord[Y] = (node[f[m][is].n].coord[Y] +
                        node[f[m][js].n].coord[Y]) / 2.0;
node[ni].coord[Z] = (node[f[m][is].n].coord[Z] +
                        node[f[m][js].n].coord[Z]) / 2.0;
if(sol_face[m].type != 0) point_projection(node[ni].coord);
set_no_intersect_node(ni);
ni++;
af[0].n = ni-2;
af[0].a = a180;
af[0].f = nf;
af[0].d = dist(node[ni-2].coord, node[f[m][im].n].coord);
w = 1;
i = im;
while(i != is){
  af[w].n = f[m][i].n;
  af[w].a = f[m][i].a;
  af[w].f = -1;
  af[w].d = f[m][i].d;
  w++;
  i = incr(i);
}
af[w].n = f[m][is].n;
af[w].f = -1;
w++;
af[w].n = ni-1;
af[w].a = a180;
af[w].f = -1;
af[w].d = dist(node[ni-1].coord, node[f[m][js].n].coord);
w++;
i = js;
while(i != jm){
  af[w].n = f[m][i].n;
  af[w].a = f[m][i].a;
  af[w].f = -1;
  af[w].d = f[m][i].d;
  w++;
  if(ao == 1) {i = dec(i); commute_face_nodes(f[m][i].n);}
  else i = inc(i);
}
af[w].n = f[m][jm].n;
af[w].f = nf;
w++;
commute_face_nodes(f[m][jm].n);
for(i=kpi+1 ; i < kj ; i++){
  af[w].n = f[m][i].n;
  af[w].a = f[m][i].a;
  af[w].f = f[m][i].f;
  af[w].d = f[m][i].d;
  w++;
}
for(i=kpj+1 ; i <= pi[m] ; i++){
  af[w].n = f[m][i].n;
  af[w].a = f[m][i].a;
```

```
    af[w].f = f[m][i].f;
    af[w].d = f[m][i].d;
    w++;
  }
  for(i=0 ; i < w ; i++){
    f[m][ki+i].n = af[i].n;
    f[m][ki+i].a = af[i].a;
    f[m][ki+i].f = af[i].f;
    f[m][ki+i].d = af[i].d;
  }
  k = kpi - ki + 1;
  kpi = kpi + (kpj - kj + 1) + 2;
  pi[m] = pi[m] + 2;
  front_node_ang_dist(nf, ki+1);
  front_node_ang_dist(nf, ki+k);
  front_node_ang_dist(nf, ki+k+2);
  front_node_ang_dist(nf, kpi);

  create_face(ni-2, ni-1, f[m][ki+k+2].n, f[m][kpi].n);
  create_face(f[m][ki+1].n, f[m][ki+k].n, ni-1, ni-2);
}

/*INTERSECT TWO FRONTS WITH ONE FACE */
void intersect_two_fronts_with_one_face(int ao, int nf,
                                        int im, int is, int jm, int js)
{
  int i, k, w;
  set_no_intersect_node(f[m][im].n);
  set_no_intersect_node(f[m][is].n);
  set_no_intersect_node(f[m][jm].n);
  set_no_intersect_node(f[m][js].n);
  w = 0;
  i = im;
  while(i != is){
    af[w].n = f[m][i].n;
    af[w].a = f[m][i].a;
    af[w].f = -1;
    af[w].d = f[m][i].d;
    w++;
    i = incr(i);
  }
  af[0].f = nf;
  af[w].n = f[m][is].n;
  af[w].f = -1;
  w++;
  i = js;
  while(i != jm){
    af[w].n = f[m][i].n;
    af[w].a = f[m][i].a;
    af[w].f = -1;
    af[w].d = f[m][i].d;
    w++;
    if(ao == 1) {i = dec(i); commute_face_nodes(f[m][i].n);}
    else i = inc(i);
  }
  af[w].n = f[m][jm].n;
  af[w].f = nf;
  w++;
  commute_face_nodes(f[m][jm].n);
  for(i=kpi+1 ; i < kj ; i++){
    af[w].n = f[m][i].n;
    af[w].a = f[m][i].a;
    af[w].f = f[m][i].f;
    af[w].d = f[m][i].d;
    w++;
  }
  for(i=kpj+1 ; i <= pi[m] ; i++){
    af[w].n = f[m][i].n;
    af[w].a = f[m][i].a;
    af[w].f = f[m][i].f;
    af[w].d = f[m][i].d;
    w++;
  }
  for(i=0 ; i < w ; i++){
    f[m][ki+i].n = af[i].n;
    f[m][ki+i].a = af[i].a;
    f[m][ki+i].f = af[i].f;
    f[m][ki+i].d = af[i].d;
  }
  k = kpi - ki + 1;
  kpi = kpi + (kpj - kj + 1);
  front_node_ang_dist(nf, ki);
  front_node_ang_dist(nf, ki+k-1);
  front_node_ang_dist(nf, ki+k);
  front_node_ang_dist(nf, kpi);
  create_face(f[m][ki].n, f[m][ki+k-1].n, f[m][ki+k].n, f[m][kpi].n);
}

/* INTERSECT TWO FRONTS WITH SEAM */
void intersect_two_fronts_with_seam(int ao, int nf,
                                    int im, int is, int jm, int js)
{
  int i, j, jj, k, w, anf;
  set_no_intersect_node(f[m][im].n);
  set_no_intersect_node(f[m][is].n);
  set_no_intersect_node(f[m][jm].n);
  set_no_intersect_node(f[m][js].n);
  node[f[m][im].n].coord[X] = (node[f[m][jm].n].coord[X]
                               + node[f[m][im].n].coord[X]) / 2.0;
  node[f[m][im].n].coord[Y] = (node[f[m][jm].n].coord[Y]
                               + node[f[m][im].n].coord[Y]) / 2.0;
  node[f[m][im].n].coord[Z] = (node[f[m][jm].n].coord[Z]
                               + node[f[m][im].n].coord[Z]) / 2.0;
  if(sol_face[m].type != 0) point_projection(node[f[m][im].n].coord);
  for(j=0 ; j < node[f[m][jm].n].nfi ; j++){
    node[f[m][im].n].nface[node[f[m][im].n].nfi]=node[f[m][jm].n].nface[j];
    node[f[m][im].n].nfi++;
    for(jj=0 ; jj < 4 ; jj++){
      if(face[node[f[m][jm].n].nface[j]].fnode[jj] == f[m][jm].n){
        face[node[f[m][jm].n].nface[j]].fnode[jj] = f[m][im].n;
        break;
      }
    }
  }
  for(i=0 ; i <= pi[m] ; i++){
    if((f[m][i].n == f[m][jm].n) && (i != jm) &&
       ((f[m][i+1].f == -1) || (f[m][i-1].f == -1))){
      j = i;
      while(f[m][j].f == -1) j = decr(j);
      anf = f[m][j].f;
      f[m][i].n = f[m][im].n;
          set_ki_kpi(anf);
      front_node_ang_dist(anf, decr(i));
      front_node_ang_dist(anf, i);
      front_node_ang_dist(anf, incr(i));
      set_ki_kpi(nf);
      continue;
    }
  }
  node[f[m][jm].n].no = -1;
  node[f[m][jm].n].coord[X] = 1E+10;
  node[f[m][jm].n].coord[Y] = 1E+10;
  node[f[m][jm].n].coord[Z] = 1E+10;
  node[f[m][is].n].coord[X] = (node[f[m][js].n].coord[X]
                              + node[f[m][is].n].coord[X]) / 2.0;
  node[f[m][is].n].coord[Y] = (node[f[m][js].n].coord[Y]
                              + node[f[m][is].n].coord[Y]) / 2.0;
  node[f[m][is].n].coord[Z] = (node[f[m][js].n].coord[Z]
                              + node[f[m][is].n].coord[Z]) / 2.0;
  if(sol_face[m].type != 0) point_projection(node[f[m][is].n].coord);
  for(j=0 ; j < node[f[m][js].n].nfi ; j++){
    node[f[m][is].n].nface[node[f[m][is].n].nfi]=node[f[m][js].n].nface[j];
    node[f[m][is].n].nfi++;
    for(jj=0 ; jj < 4 ; jj++){
      if(face[node[f[m][js].n].nface[j]].fnode[jj] == f[m][js].n){
        face[node[f[m][js].n].nface[j]].fnode[jj] = f[m][is].n;
        break;
      }
    }
  }
  for(i=0 ; i <= pi[m] ; i++){
    if((f[m][i].n == f[m][js].n) && (i != js) &&
       ((f[m][i+1].f == -1) || (f[m][i-1].f == -1))){
      j = i;
      while(f[m][j].f == -1) j = decr(j);
      anf = f[m][j].f;
      f[m][i].n = f[m][is].n;
      set_ki_kpi(anf);
      front_node_ang_dist(anf, decr(i));
      front_node_ang_dist(anf, i);
      front_node_ang_dist(anf, incr(i));
      set_ki_kpi(nf);
      continue;
    }
  }
  node[f[m][js].n].no = -1;
  node[f[m][js].n].coord[X] = 1E+10;
  node[f[m][js].n].coord[Y] = 1E+10;
  node[f[m][js].n].coord[Z] = 1E+10;
  w = 0;
  i = im;
  while(i != is){
    af[w].n = f[m][i].n;
    af[w].a = f[m][i].a;
    af[w].f = -1;
    af[w].d = f[m][i].d;
    w++;
    i = incr(i);
  }
  af[0].f = nf;
  af[w].n = f[m][is].n;
  af[w].f = -1;
  w++;

  if(ao == 1) i = dec(js);
  else i = inc(js);
  while(i != jm){
    af[w].n = f[m][i].n;
    af[w].a = f[m][i].a;
    af[w].f = -1;
    af[w].d = f[m][i].d;
    w++;
    if(ao == 1) {i = dec(i); commute_face_nodes(f[m][i].n);}
    else i = inc(i);
  }
  af[w-1].f = nf;
  commute_face_nodes(f[m][jm].n);
  for(i=kpi+1 ; i < kj ; i++){
    af[w].n = f[m][i].n;
    af[w].a = f[m][i].a;
    af[w].f = f[m][i].f;
    af[w].d = f[m][i].d;
    w++;
  }
  for(i=kpj+1 ; i <= pi[m] ; i++){
    af[w].n = f[m][i].n;
    af[w].a = f[m][i].a;
    af[w].f = f[m][i].f;
    af[w].d = f[m][i].d;
    w++;
  }
  for(i=0 ; i < w ; i++){
    f[m][ki+i].n = af[i].n;
    f[m][ki+i].a = af[i].a;
    f[m][ki+i].f = af[i].f;
    f[m][ki+i].d = af[i].d;
  }
  k = kpi - ki + 1;
  kpi = kpi + (kpj - kj + 1) - 2;
  pi[m] = pi[m] - 2;
  front_node_ang_dist(nf, ki);
  front_node_ang_dist(nf, ki+1);
  front_node_ang_dist(nf, ki+k-2);
  front_node_ang_dist(nf, ki+k-1);
  front_node_ang_dist(nf, ki+k);
  front_node_ang_dist(nf, kpi);
}
```

A.15 Function *intersection(int nf, int nff)*

```
int intersection(int nf, int nff)
{
  int i, j, k, q=0, w, ie, ao, im, jm, is, js, ia, ja, in=0, ins=0, a;
  int  odd, odds;
  REAL d, dm, di, dj, ds, aux;
  POINT P;
  ap_Param parm;
  printf("   intersection = ");
  if(nf == nff){
    d = 20.0 * msize[m];
    ds = 20.0 * msize[m];
    for(i=ki ; i < kpi-3 ; i++){
      j = i + 4;
      if(q == 0){
        ie = decr(decr(decr(i)));
        if(ie == ki) q = 1;
      }
      ao = 0;
      while(j != ie){
        a = 0;
        ao = ((ao+1) & 1);
        aux = dist(node[f[m][i].n].coord, node[f[m][j].n].coord);
        if(aux < d) {d = aux; im = i; jm = j; odd = ao;}

        if((ins == 0) || (aux < ds)){
          di = (f[m][decr(i)].d + f[m][i].d) / 2.0;
          dj = (f[m][decr(j)].d + f[m][j].d) / 2.0;

          k = incr(incr(j));
          if((incr(incr(k)) == i) && (f[m][k].a <= a60)) a = 1;

          k = i + 2;
          if((k+2 == j) && (f[m][k].a <= a60)) a = 1;

          if(((a == 0) && (aux < 1.05*(di+dj))) ||
             ((a == 1) && (aux < 1.05*5.0*(di+dj)/8.0))){
```

```
          if(check_no_intersect_node(-1, i, j) == 0)
            {ds = aux; is = i; js = j; odds = ao; ins = 1;}
        }
      }
      j = incr(j);
    }
  }
  if(check_no_intersect_node(nf, im, jm) == 0){
    di = (f[m][decr(im)].d + f[m][im].d) / 2.0;
    dj = (f[m][decr(jm)].d + f[m][jm].d) / 2.0;
    k = incr(incr(jm));
    j = incr(incr(k));
    if((j == im) && (f[m][k].a <= a60)) in = -1;
    i = im + 4;
    k = im + 2;
    if((i == jm) && (f[m][k].a <= a60)) in = -1;
    dm = di + dj;
    if((in == 0) && (d < 1.05*dm)){
      if(odd == 0) in = 2;
      if(odd == 1){
        if(d < dm/2.0){
          if(rect_face == 1) in = 1;
          if(rect_face == 0) in = 3;
        }
        else in = 1;
      }
    }
    if(in == -1){
      if(d < 1.05*5.0*dm/8.0) in = 1;
      else in = 0;
    }
  }
  if((in == 0) && (ins == 1) && (check_no_intersect_node(nf, is, js) == 0)){
    im = is;
    jm = js;
    if(odds == 0) in = 2;
    if(odds == 1){
      di = (f[m][decr(im)].d + f[m][im].d) / 2.0;
      dj = (f[m][decr(jm)].d + f[m][jm].d) / 2.0;
      dm = di + dj;
      if(ds < dm/2.0){
        if(rect_face == 1) in = 1;
        if(rect_face == 0) in = 3;
      }
      else in = 1;
    }
  }
  if(in != 0){
    if(in == 1){
      printf("intersect_with_new_node  ");
      intersect_with_new_node(nf, im, jm);
      return 1;
    }
    if(in == 2){
        printf("intersect_move_nodes   ");
        intersect_move_nodes(d, nf, im, jm);
        if(repr == 1) reprint_face(pff[nf]);
        return 1;
      }
      if(in == 3){
        printf("intersect_seam_nodes   ");
        intersect_seam_nodes(nf, im, jm);
        if(repr == 1) reprint_face(pff[nf]);
        return 1;
      }
    }
    return 0;
  }
  if(nf != nff){
    set_ki_kpi(nf);
    kj = 0;
    kpj = 0;
    for(i=0 ; i <= pi[m] ; i++){
      if(f[m][i].f == nff){
        kj = i;
        i++;
        for(j=i ; j <= pi[m] ; j++) if(f[m][j].f == nff) {kpj = j;  break;}
        break;
      }
    }
    if((kpj-kj) < 2) return 0;
    d = 20.0 * msize[m];
    for(i=ki ; i <= kpi ; i++){
      for(j=kj ; j <= kpj ; j++){
        aux = dist(node[f[m][i].n].coord, node[f[m][j].n].coord);
        if(aux < d) {d = aux; im = i; jm = j;}
      }
    }
    P[X] = (node[f[m][im].n].coord[X] + node[f[m][jm].n].coord[X]) / 2.0;
    P[Y] = (node[f[m][im].n].coord[Y] + node[f[m][jm].n].coord[Y]) / 2.0;
    P[Z] = (node[f[m][im].n].coord[Z] + node[f[m][jm].n].coord[Z]) / 2.0;
    if(sol_face[m].type != 0) point_projection(P);
    if(ap_facept2parm(sol_id, sol_face[m].id, P, &parm) != AP_NORMAL) return 0;
    if(kj < ki){
      k = jm;  jm  = im;  im  = k;
      k = kj;  kj  = ki;  ki  = k;
      k = kpj; kpj = kpi; kpi = k;
      k = nff; nff = nf;  nf  = k;
    }
    is = decr(im);
    ia = incr(im);
    js = dec(jm);
    ja = inc(jm);
    ao = 1;
    if(turn[m][nf] == turn[m][nff]) {js = inc(jm); ja = dec(jm); ao = -1;}
    if(dist(node[f[m][ia].n].coord, node[f[m][ja].n].coord) <
       dist(node[f[m][is].n].coord, node[f[m][js].n].coord))
      {is = im; im = ia; js = jm; jm = ja;}
    di = (f[m][is].d + f[m][im].d) / 2.0;
    dj = (f[m][js].d + f[m][jm].d) / 2.0;
    dm = di + dj;
    if((0.75*dm < d) && (d < 1.05*dm)){
      printf("intersect_two_fronts_with_two_new_nodes   ");
      intersect_two_fronts_with_two_new_nodes(ao, nf, im, is, jm, js);
      if(pff[nf] > pff[nff]) pff[nf] = pff[nff];
      set_origin(nf, nff);
      return 1;
    }
    if((dm/3.0 <= d) && (d <= 0.75*dm)){
      printf("intersect_two_fronts_with_one_face   ");
      intersect_two_fronts_with_one_face(ao, nf, im, is, jm, js);
      if(pff[nf] > pff[nff]) pff[nf] = pff[nff];
      set_origin(nf, nff);
      return 1;
    }
    if(d < dm/3.0){
      printf("intersect_two_fronts_with_seam   ");
      intersect_two_fronts_with_seam(ao, nf, im, is, jm, js);
      if(pff[nf] > pff[nff]) pff[nf] = pff[nff];
      set_origin(nf, nff);
      if(repr == 1) reprint_face(pff[nf]);
      return 1;
    }
    return 0;
  }
}
```

A.16 Function *close_front(int nf)*

```
void close_front(int nf)
{
  int i, j, jj, k, i0, i1, j0, j1, km=kpi, km3, s=0, anf;
  printf("   close_front\n");
  if((kpi-ki+1) == 4){
    for(i=ki ; i <= ki+1 ; i++){
      j = i + 2;
      i1 = i + 1;
      i0 = decr(i);
      if((f[m][i].a <= a45) && (f[m][j].a <= a45)){s = 1; break;}
    }
    if(s == 0){
      create_face(f[m][kpi].n, f[m][kpi-1].n, f[m][ki+1].n, f[m][ki].n);
      f[m][ki+1].f = nf;
      f[m][kpi-1].f = nf;
    }
    else{
      node[f[m][i0].n].coord[X] = (node[f[m][i1].n].coord[X]
                                  + node[f[m][i0].n].coord[X]) / 2.0;
      node[f[m][i0].n].coord[Y] = (node[f[m][i1].n].coord[Y]
                                + node[f[m][i0].n].coord[Y]) / 2.0;
      node[f[m][i0].n].coord[Z] = (node[f[m][i1].n].coord[Z]
                                + node[f[m][i0].n].coord[Z]) / 2.0;
      if(sol_face[m].type != 0) point_projection(node[f[m][i0].n].coord);
      for(j=0 ; j < node[f[m][i1].n].nfi ; j++){
        node[f[m][i0].n].nface[node[f[m][i0].n].nfi]=node[f[m][i1].n].nface[j];
        node[f[m][i0].n].nfi++;
        for(jj=0 ; jj < 4 ; jj++){
          if(face[node[f[m][i1].n].nface[j]].fnode[jj] == f[m][i1].n){
            face[node[f[m][i1].n].nface[j]].fnode[jj] = f[m][i0].n;
            break;
          }
        }
      }
      node[f[m][i1].n].no = -1;
      node[f[m][i1].n].coord[X] = 1E+10;
      node[f[m][i1].n].coord[Y] = 1E+10;
      node[f[m][i1].n].coord[Z] = 1E+10;

      for(i=ki+1 ; i < kpi ; i++) f[m][i].f = nf;
      if(repr == 1) reprint_face(pff[nf]);
    }
  }
  if((kpi-ki+1) > 4){
    for(i=ki ; i <= ki+2 ; i++){
      j = i + 3;
      i1 = i + 1;
      i0 = decr(i);
      j1 = j - 1;
      j0 = incr(j);
      if((f[m][i].a <= a90) && (f[m][j].a <= a90) &&
         (f[m][i0].a >= a120) && (f[m][j0].a >= a120) &&
         (f[m][i1].a >= a120) && (f[m][j1].a >= a120)) {s = 1; break;}
    }
    if(s == 0){
      km3 = -1;
      for(i=ki ; i <= kpi ; i++)
        if((node[f[m][i].n].nfi < 2) && (f[m][i].a > a135))
          {km = i; km3 = incr(incr(incr(i)));}
      if(km3 == -1)
        for(i=ki ; i <= kpi ; i++){
          j = incr(incr(incr(i)));
          if((f[m][km].a <= f[m][i].a) && (node[f[m][i].n].nfi < 3) &&
             (node[f[m][j].n].nfi < 3)) {km = i; km3 = j;}
        }
      if(km3 == -1)
        for(i=ki ; i <= kpi ; i++){
          j = incr(incr(incr(i)));
          if((f[m][km].a <= f[m][i].a) && (node[f[m][i].n].nfi < 4) &&
             (node[f[m][j].n].nfi < 4)) {km = i; km3 = j;}
        }
      if(km3 == -1)
        for(i=ki ; i <= kpi ; i++)
      if(f[m][km].a <= f[m][i].a) {km = i; km3 = incr(incr(incr(km)));}
    create_face(f[m][km3].n, f[m][decr(km3)].n, f[m][incr(km)].n,f[m][km].n);
    create_face(f[m][km3].n, f[m][km].n, f[m][decr(km)].n,f[m][incr(km3)].n);
    for(i=ki+1 ; i < kpi ; i++) f[m][i].f = nf;
  }
  else{
    node[f[m][i0].n].coord[X] = (node[f[m][i1].n].coord[X] +
                                node[f[m][i0].n].coord[X]) / 2.0;
    node[f[m][i0].n].coord[Y] = (node[f[m][i1].n].coord[Y] +
                                node[f[m][i0].n].coord[Y]) / 2.0;
    node[f[m][i0].n].coord[Z] = (node[f[m][i1].n].coord[Z] +
                                node[f[m][i0].n].coord[Z]) / 2.0;
    if(sol_face[m].type != 0) point_projection(node[f[m][i0].n].coord);
    for(j=0 ; j < node[f[m][i1].n].nfi ; j++){
      node[f[m][i0].n].nface[node[f[m][i0].n].nfi]=node[f[m][i1].n].nface[j];
      node[f[m][i0].n].nfi++;
      for(jj=0 ; jj < 4 ; jj++){
        if(face[node[f[m][i1].n].nface[j]].fnode[jj] == f[m][i1].n){
          face[node[f[m][i1].n].nface[j]].fnode[jj] = f[m][i0].n;
          break;
        }
      }
    }
    node[f[m][j0].n].coord[X] = (node[f[m][j1].n].coord[X] +
                                node[f[m][j0].n].coord[X]) / 2.0;
    node[f[m][j0].n].coord[Y] = (node[f[m][j1].n].coord[Y] +
                                node[f[m][j0].n].coord[Y]) / 2.0;
    node[f[m][j0].n].coord[Z] = (node[f[m][j1].n].coord[Z] +
                                node[f[m][j0].n].coord[Z]) / 2.0;
    if(sol_face[m].type != 0) point_projection(node[f[m][j0].n].coord);
    for(j=0 ; j < node[f[m][j1].n].nfi ; j++){
      node[f[m][j0].n].nface[node[f[m][j0].n].nfi]=node[f[m][j1].n].nface[j];
      node[f[m][j0].n].nfi++;
      for(jj=0 ; jj < 4 ; jj++){
        if(face[node[f[m][j1].n].nface[j]].fnode[jj] == f[m][j1].n){
          face[node[f[m][j1].n].nface[j]].fnode[jj] = f[m][j0].n;
          break;
        }
      }
    }
    for(i=ki+1 ; i < kpi ; i++) f[m][i].f = nf;
    for(i=0 ; i <= pi[m] ; i++){
      if((f[m][i].n == f[m][i1].n) && (i != i1) &&
         ((f[m][incr(i)].f == -1) || (f[m][decr(i)].f == -1))){
        j = i;
        while(f[m][j].f == -1) j = decr(j);
        anf = f[m][j].f;
        f[m][i].n = f[m][i0].n;
        set_ki_kpi(anf);
        front_node_ang_dist(anf, decr(i));
        front_node_ang_dist(anf, i);
        front_node_ang_dist(anf, incr(i));
        set_ki_kpi(nf);
        continue;
        }
        if((f[m][i].n == f[m][j1].n) && (i != j1) &&
           ((f[m][incr(i)].f == -1) || (f[m][decr(i)].f == -1))){
          j = i;
          while(f[m][j].f == -1) j = decr(j);
          anf = f[m][j].f;
          f[m][i].n = f[m][j0].n;
          set_ki_kpi(anf);
          front_node_ang_dist(anf, decr(i));
          front_node_ang_dist(anf, i);
          front_node_ang_dist(anf, incr(i));
          set_ki_kpi(nf);
        }
      }
      node[f[m][i1].n].no = -1;
      node[f[m][i1].n].coord[X] = 1E+10;
      node[f[m][i1].n].coord[Y] = 1E+10;
      node[f[m][i1].n].coord[Z] = 1E+10;
      node[f[m][j1].n].no = -1;
      node[f[m][j1].n].coord[X] = 1E+10;
      node[f[m][j1].n].coord[Y] = 1E+10;
      node[f[m][j1].n].coord[Z] = 1E+10;
      if(repr == 1) reprint_face(pff[nf]);
    }
  }
}
```

A.17 Function *smooth_front(int nf)*

```
int smooth_front(int nf)
{
  int i, k, i0, i2, in=0;
  REAL d, tol;
  POINT v;
  printf("   smooth_front = ");
  tol = size / 20.0;
  for(k=0 ; k < 2 ; k++){
    for(i=ki ; i <= kpi ; i++){
      i0 = decr(i);
      i2 = incr(i);
      d = (f[m][i0].d + f[m][i].d) / 2.0;
      if((f[m][i].d-d) > tol){
        in = 1;
        vector(i2, i, v);
        scalar_vector(d/f[m][i].d, v);
        point_vector(i2, v, node[f[m][i].n].coord);
        if(sol_face[m].type != 0) point_projection(node[f[m][i].n].coord);
        front_node_ang_dist(nf, i0);
        front_node_ang_dist(nf, i);
        front_node_ang_dist(nf, i2);
      }
      if((f[m][i0].d-d) > tol){
        in = 1;
        vector(i0, i, v);
        scalar_vector(d/f[m][i0].d, v);
        point_vector(i0, v, node[f[m][i].n].coord);
        if(sol_face[m].type != 0) point_projection(node[f[m][i].n].coord);
        front_node_ang_dist(nf, i0);
        front_node_ang_dist(nf, i);
        front_node_ang_dist(nf, i2);
      }
    }
  }
  if((in == 1) && (repr ==1)) reprint_face(pff[nf]);
  return in;
}
```

A.18 Function *smooth()*

```
void smooth()
{
  int i, j, k, ii, i0, i1, an, as;
  POINT P;
  printf("   smooth\n");
  if(sol_face[m].type == 0){as = (fi - pf) / 5; if(as < 3) as = 3;}
  else as = 2;
  for(ii=0 ; ii < as ; ii++){
    for(j=pn ; j < ni ; j++){
      if(node[j].no != -1){
        an = 0;
        P[X] = 0.0;
        P[Y] = 0.0;
        P[Z] = 0.0;
        for(k=0 ; k < node[j].nfi ; k++){
          for(i=0 ; i < 4 ; i++) if(face[node[j].nface[k]].fnode[i]== j) break;
          i0 = dec4(i);
          i1 = inc4(i);
          P[X] = P[X] + node[face[node[j].nface[k]].fnode[i0]].coord[X];
          P[Y] = P[Y] + node[face[node[j].nface[k]].fnode[i0]].coord[Y];
          P[Z] = P[Z] + node[face[node[j].nface[k]].fnode[i0]].coord[Z];
          P[X] = P[X] + node[face[node[j].nface[k]].fnode[i1]].coord[X];
          P[Y] = P[Y] + node[face[node[j].nface[k]].fnode[i1]].coord[Y];
          P[Z] = P[Z] + node[face[node[j].nface[k]].fnode[i1]].coord[Z];
          an = an + 2;
        }
        node[j].coord[X] = P[X] / an;
        node[j].coord[Y] = P[Y] / an;
        node[j].coord[Z] = P[Z] / an;
        if(sol_face[m].type != 0) point_projection(node[j].coord);
      }
    }
  }
  if(repr == 1) reprint_face(pf);
}
```

A.19 Function *clean_up()*

```
void clean_up()
{
  int i, j, k, k1, k0, f0, f1;

  printf("   clean_up\n");

  for(i=pn ; i < ni ; i++){
    if((node[i].no != -1) && (node[i].nfi == 2)){
      f0 = node[i].nface[0];
      f1 = node[i].nface[1];

      for(k=0 ; k < 4 ; k++) if(face[f0].fnode[k] == i) break;
      k0 = decr(k);
      k1 = incr(k);
      for(j=0 ; j < 4 ; j++) if(face[f1].fnode[j] == i) break;

      face[f0].fnode[k] = face[f1].fnode[inc4(inc4(j))];
      face[f1].no = -1;

      node[i].no = -1;
      node[i].coord[X] = 1E+10;
      node[i].coord[Y] = 1E+10;
      node[i].coord[Z] = 1E+10;

      for(j=0 ; j < node[face[f0].fnode[k]].nfi ; j++){
        if(node[face[f0].fnode[k]].nface[j] == f1){
          node[face[f0].fnode[k]].nface[j] = f0;
          break;
        }
      }

      for(j=0 ; j < node[face[f0].fnode[k0]].nfi ; j++){
        if(node[face[f0].fnode[k0]].nface[j] == f1){
          k = j;
          break;
        }
      }
      for(j=0 ; j < (node[face[f0].fnode[k0]].nfi-1) ; j++){
        node[face[f0].fnode[k0]].nface[j]= node[face[f0].fnode[k0]].nface[j+1];
      }
      node[face[f0].fnode[k0]].nfi--;

      for(j=0 ; j < node[face[f0].fnode[k1]].nfi ; j++){
        if(node[face[f0].fnode[k1]].nface[j] == f1){
          k = j;
          break;
        }
      }
      for(j=0 ; j < (node[face[f0].fnode[k1]].nfi-1) ; j++){
        node[face[f0].fnode[k1]].nface[j]= node[face[f0].fnode[k1]].nface[j+1];
      }
      node[face[f0].fnode[k1]].nfi--;
    }
  }
  if(repr == 1) reprint_face(pf);
}
```

A.20 Function *initial_3D_front()*

```
/* SET ADJACENT FACES */
void set_adjacent_faces(int i)
{
  int j, k;
  for(j=0 ; j < node[face[i].fnode[0]].nfi ; j++){
    if(face[node[face[i].fnode[0]].nface[j]].no != -1){
      if(node[face[i].fnode[0]].nface[j] != i){
        for(k=0 ; k < 4 ; k++){
          if(face[node[face[i].fnode[0]].nface[j]].fnode[k] ==
             face[i].fnode[1]){
            face[i].fface[0] = node[face[i].fnode[0]].nface[j];
            break;
          }
          if(face[node[face[i].fnode[0]].nface[j]].fnode[k] ==
             face[i].fnode[3]){
            face[i].fface[3] = node[face[i].fnode[0]].nface[j];
            break;
          }
        }
      }
    }
  }
  for(j=0 ; j < node[face[i].fnode[2]].nfi ; j++){
    if(face[node[face[i].fnode[2]].nface[j]].no != -1){
      if(node[face[i].fnode[2]].nface[j] != i){
        for(k=0 ; k < 4 ; k++){
          if(face[node[face[i].fnode[2]].nface[j]].fnode[k] ==
             face[i].fnode[1]){
            face[i].fface[1] = node[face[i].fnode[2]].nface[j];
            break;
          }
          if(face[node[face[i].fnode[2]].nface[j]].fnode[k] ==
             face[i].fnode[3]){
            face[i].fface[2] = node[face[i].fnode[2]].nface[j];
            break;
          }
        }
      }
    }
  }
}

/* INITIAL 3D−FRONT */
void initial_3D_front()
{
  int i, w;
  REAL a, d;
  for(i=0 ; i < fi ; i++){
    if(face[i].no != −1){
      if(sol_face[face[i].fm].type != 0)
        ap_pt_norm2face(sol_id, sol_face[face[i].fm].id,
                  node[face[i].fnode[1]].coord, TRUE, n[face[i].fm]);

      face_norm(face[i].fnode[0], face[i].fnode[1], face[i].fnode[2],
              face[i].fnode[3], face[i].norm);

      a = inner_product(face[i].norm, n[face[i].fm]);
      if(a > 0){
        scalar_vector(−1.0, face[i].norm);
        w = face[i].fnode[1];
        face[i].fnode[1] = face[i].fnode[3];
        face[i].fnode[3] = w;
      }
      set_adjacent_faces(i);
    }
    else face[i].ord = −1;
  }
}
```

A.21 Function *face_angles(int f, int kf)*

```
REAL face_angles(int f, int kf)
{
  int k, k1, k2, k3;
  REAL d, d0, d1, dp, a;
  POINT v, v0, v1, P, p0, p1;
  k1 = inc4(kf);
  vector_no(face[f].fnode[kf], face[f].fnode[k1], v);
  k2 = inc4(k1);
  k3 = inc4(k2);
  p0[X] = (node[face[f].fnode[kf]].coord[X] +
          node[face[f].fnode[k1]].coord[X]) / 2.0;
  p0[Y] = (node[face[f].fnode[kf]].coord[Y] +
          node[face[f].fnode[k1]].coord[Y]) / 2.0;
  p0[Z] = (node[face[f].fnode[kf]].coord[Z] +
          node[face[f].fnode[k1]].coord[Z]) / 2.0;
  p1[X] = (node[face[f].fnode[k2]].coord[X] +
          node[face[f].fnode[k3]].coord[X]) / 2.0;
  p1[Y] = (node[face[f].fnode[k2]].coord[Y] +
          node[face[f].fnode[k3]].coord[Y]) / 2.0;
  p1[Z] = (node[face[f].fnode[k2]].coord[Z] +
          node[face[f].fnode[k3]].coord[Z]) / 2.0;
  v0[X] = p1[X] - p0[X];
  v0[Y] = p1[Y] - p0[Y];
  v0[Z] = p1[Z] - p0[Z];
  vector_product(v0, v, P);
  vector_product(v, P, v0);
  d0 = vector_norm(v0);
  for(k=0 ; k < 4 ; k++)
    if(face[face[f].fface[kf]].fnode[k] == face[f].fnode[kf]) break;
  k2 =inc4(inc4(k));
  if(face[face[f].fface[kf]].fnode[inc4(k)] == face[f].fnode[k1]) k3 = dec4(k);
  else k3 = inc4(k);
  p1[X] = (node[face[face[f].fface[kf]].fnode[k2]].coord[X] +
          node[face[face[f].fface[kf]].fnode[k3]].coord[X]) / 2.0;
  p1[Y] = (node[face[face[f].fface[kf]].fnode[k2]].coord[Y] +
          node[face[face[f].fface[kf]].fnode[k3]].coord[Y]) / 2.0;
  p1[Z] = (node[face[face[f].fface[kf]].fnode[k2]].coord[Z] +
          node[face[face[f].fface[kf]].fnode[k3]].coord[Z]) / 2.0;
  v1[X] = p1[X] - p0[X];
  v1[Y] = p1[Y] - p0[Y];
  v1[Z] = p1[Z] - p0[Z];
  vector_product(v1, v, P);
  vector_product(v, P, v1);
  d1 = vector_norm(v1);
  scalar_vector(d0/d1, v1);
  sub_vector(v0, v1, p0);
  vector_product(p0, v, p1);
  dp = vector_norm(p1);
  add_vector(face[f].norm, face[face[f].fface[kf]].norm, v);
  d = vector_norm(v);
  a = inner_product(p1, v) / (dp * d);
  if(a < 0) scalar_vector(-1.0, p1);
  a = inner_product(p1, v0) / (dp * d0);
  a = 2 * acos(a);
  return a;
}
```

A.22 Auxiliary functions to advance the 3D-front

```
/* PROJECTION */
int projection(int f, int n0)
{
  int i, j, k, k0, k1, w=0, no[15], i0, i1;
  REAL d, dv, a;
  POINT v;
  for(i=0 ; i < 15 ; i++) no[i] = -1;
  for(i=0 ; i < node[n0].nfi ; i++){
    if(face[node[n0].nface[i]].no != -1){
      for(k=0 ; k < 4 ; k++) if(face[node[n0].nface[i]].fnode[k] == n0) break;
      k0 = dec4(k);
      k1 = inc4(k);
      i0 = 0;
      i1 = 0;
      for(j=0 ; j < w ; j++){
        if(no[j] == face[node[n0].nface[i]].fnode[k0]) i0 = 1;
        if(no[j] == face[node[n0].nface[i]].fnode[k1]) i1 = 1;
      }
      if(i0 == 0){
        no[w] = face[node[n0].nface[i]].fnode[k0];
        w++; if(w > 15)printf("     ERROR-1\n");
      }
      if(i1 == 0){
        no[w] = face[node[n0].nface[i]].fnode[k1];
        w++; if(w > 15)printf("     ERROR-2\n");
      }
    }
  }
  for(i=0 ; i < w ; i++){
    vector_no(n0, no[i], v);
    dv = vector_norm(v);
    a = inner_product(face[f].norm, v) / dv;
    if(0.7 < a) return no[i];
  }
  d = 0.0;
  for(i=0 ; i < w ; i++) d = d + dist(node[n0].coord, node[no[i]].coord);
  d = d / w;
  v[X] = 0.0;
  v[Y] = 0.0;
  v[Z] = 0.0;
  for(i=0 ; i < node[n0].nfi ; i++){
    if(face[node[n0].nface[i]].no != -1){
      v[X] = v[X] + face[node[n0].nface[i]].norm[X];
      v[Y] = v[Y] + face[node[n0].nface[i]].norm[Y];
      v[Z] = v[Z] + face[node[n0].nface[i]].norm[Z];
    }
  }
  dv = vector_norm(v);
  scalar_vector(d/dv, v);
  node[ni].no = ni;
  point_vector_no(n0, v, node[ni].coord);
  ni++;
  return (ni-1);
}

/* NEW FACE */
void new_face(int nf, int n1, int n2, int n3, int n4)
{
  face[fi].ord = nf;
  face[fi].no = fi;
  face[fi].fnode[0] = n1;
  face[fi].fnode[1] = n2;
  face[fi].fnode[2] = n3;
  face[fi].fnode[3] = n4;
  node[n1].nface[node[n1].nfi] = fi;
  node[n1].nfi++;
  node[n2].nface[node[n2].nfi] = fi;
  node[n2].nfi++;
  node[n3].nface[node[n3].nfi] = fi;
  node[n3].nfi++;
  node[n4].nface[node[n4].nfi] = fi;
  node[n4].nfi++;
  face_norm(n1, n2, n3, n4, face[fi].norm);
  fi++;
  afi++;
  lafi++;
}

/* SMOOTH ONE FACE OF THE ELEMENT */
void smooth_one_face(int c, int j)
{
  int i, k;
  REAL a, b, dv, dt, d, h;
  POINT v, t, p;
  vector_no(element[ci].enode[j], element[ci].enode[j+4], v);
  vector_no(element[ci].enode[j], element[ci].enode[inc4(j)], t);
  dv = vector_norm(v);
  dt = vector_norm(t);
  a = inner_product(v, t) / (dv * dt);
  a = acos(a);
  vector_no(element[ci].enode[i=inc4(c)], element[ci].enode[i+4], v);
  vector_no(element[ci].enode[i], element[ci].enode[c], t);
  h = vector_norm(v);
  dt = vector_norm(t);
  b = inner_product(v, t) / (h * dt);
  b = acos(b);
  vector_no(element[ci].enode[i+4], element[ci].enode[c+4], t);
  dt = vector_norm(t);
  if(a >= b){
    a = b - a;
    b = inner_product(v, t) / (h * dt);
    b = acos(b);
    b = a180 + a - b;
  }
  else{
    a = b - a;
    b = inner_product(v, t) / (h * dt);
    b = a180 - acos(b);
    b = a180 - (a + b);
  }
  d = h * sin(a) / sin(b);
  scalar_vector(d/dt, t);
  add_vector(v, t, p);
  h = vector_norm(p);
  scalar_vector(dv/h, p);
  point_vector_no(element[ci].enode[i], p, node[element[ci].enode[i+4]].coord);
  vector_no(element[ci].enode[k=inc4(j)], element[ci].enode[k+4], v);
  vector_no(element[ci].enode[k], element[ci].enode[j], t);
  dv = vector_norm(v);
  dt = vector_norm(t);
  a = inner_product(v, t) / (dv * dt);
  a = acos(a);
  vector_no(element[ci].enode[c], element[ci].enode[c+4], v);
  vector_no(element[ci].enode[c], element[ci].enode[i], t);
  h = vector_norm(v);
  dt = vector_norm(t);
  b = inner_product(v, t) / (h * dt);
  b = acos(b);
  vector_no(element[ci].enode[c+4], element[ci].enode[i+4], t);
  dt = vector_norm(t);
  if(a >= b){
    a = b - a;
    b = inner_product(v, t) / (h * dt);
    b = acos(b);
  }
  else{
    a = b - a;
    b = inner_product(v, t) / (h * dt);
    b = a180 - acos(b);
  }
  b = a180 - (a + b);
  d = h * sin(a) / sin(b);
  scalar_vector(d/dt, t);
  add_vector(v, t, p);
  h = vector_norm(p);
  scalar_vector(dv/h, p);
  point_vector_no(element[ci].enode[c], p, node[element[ci].enode[c+4]].coord);
}

/* SMOOTH TWO FACES OF THE ELEMENT */
void smooth_two_faces(int c, int j)
{
  int i, k;
  REAL a, b, dv, dt, d, h, hm;
  POINT v, t, p;
  hm = (dist(node[element[ci].enode[j]].coord,
             node[element[ci].enode[j+4]].coord) +
        dist(node[element[ci].enode[c]].coord,
             node[element[ci].enode[c+4]].coord)) / 2.0;
  vector_no(element[ci].enode[j], element[ci].enode[j+4], v);
  vector_no(element[ci].enode[j], element[ci].enode[inc4(j)], t);
  dv = vector_norm(v);
  dt = vector_norm(t);
  a = inner_product(v, t) / (dv * dt);
  a = acos(a);
  vector_no(element[ci].enode[i=inc4(c)], element[ci].enode[i+4], v);
vector_no(element[ci].enode[i], element[ci].enode[c], t);
h = vector_norm(v);
dt = vector_norm(t);
b = inner_product(v, t) / (h * dt);
b = acos(b);
vector_no(element[ci].enode[i+4], element[ci].enode[c+4], t);
dt = vector_norm(t);
if(a >= b){
  a = b - a;
  b = inner_product(v, t) / (h * dt);
  b = acos(b);
  b = a180 + a - b;
}
else{
  a = b - a;
  b = inner_product(v, t) / (h * dt);
  b = a180 - acos(b);
  b = a180 - (a + b);
}
d = h * sin(a) / sin(b);
scalar_vector(d/dt, t);
add_vector(v, t, p);
h = vector_norm(p);
scalar_vector(hm/h, p);
point_vector_no(element[ci].enode[i], p, node[element[ci].enode[i+4]].coord);
vector_no(element[ci].enode[c], element[ci].enode[c+4], v);
vector_no(element[ci].enode[c], element[ci].enode[inc4(j)], t);
dv = vector_norm(v);
dt = vector_norm(t);
a = inner_product(v, t) / (dv * dt);
a = acos(a);
vector_no(element[ci].enode[i=inc4(c)], element[ci].enode[i+4], v);
vector_no(element[ci].enode[i], element[ci].enode[j], t);
h = vector_norm(v);
dt = vector_norm(t);
b = inner_product(v, t) / (h * dt);
b = acos(b);
vector_no(element[ci].enode[i+4], element[ci].enode[j+4], t);
dt = vector_norm(t);
if(a >= b){
  a = b - a;
  b = inner_product(v, t) / (h * dt);
  b = acos(b);
  b = a180 + a - b;
}
else{
  a = b - a;
  b = inner_product(v, t) / (h * dt);
  b = a180 - acos(b);
  b = a180 - (a + b);
}
d = h * sin(a) / sin(b);
scalar_vector(d/dt, t);
add_vector(v, t, p);
```

```
  h = vector_norm(p);
  scalar_vector(hm/h, p);
  point_vector_no(element[ci].enode[i], p, node[element[ci].enode[i+4]].coord);
}

/*JOINT TWO NODES */
void joint_two_nodes(int n0, int p0)
{
  int i, k;
  for(i=0 ; i < node[n0].nfi ; i++){
    node[p0].nface[node[p0].nfi] = node[n0].nface[i];
    node[p0].nfi++;
    for(k=0 ; k < 4 ; k++){
      if(face[node[n0].nface[i]].fnode[k] == n0){
        face[node[n0].nface[i]].fnode[k] = p0;
        break;
      }
    }
  }
  for(i=0 ; i < ci ; i++){
    for(k=0 ; k < 8 ; k++){
      if(element[i].enode[k] == n0) element[i].enode[k] = p0;
    }
  }
  node[n0].no = -1;
  node[n0].coord[X] = 1E+10;
  node[n0].coord[Y] = 1E+10;
  node[n0].coord[Z] = 1E+10;
}

/* CREATE ELEMENT */
void create_element(int nf, int f)
{
  int i, j, k, p[8], v[8], nk[4], c, top=0, no=-1, cf[4]={0, 0, 0, 0};
  REAL aux;
  lafi = 0;
  for(k=0 ; k < 4 ; k++) element[ci].enode[k] = face[f].fnode[k];
  for(k=0 ; k < 4 ; k++){
    for(j=0 ; j < 4 ; j++){
      if(face[face[f].fface[k]].fnode[j] == face[f].fnode[k]) break;
    }
    if(face[face[f].fface[k]].fnode[inc4(j)] == face[f].fnode[inc4(k)]){
      p[2*k] = face[face[f].fface[k]].fnode[j=dec4(j)];
      p[2*k+1] = face[face[f].fface[k]].fnode[j=dec4(j)];
      nk[k] = j;
      for(i=0 ; i <  4 ; i++){
        if(face[face[face[f].fface[k]].fface[j]].fnode[i] == p[2*k+1]) break;
      }
      if(face[face[face[f].fface[k]].fface[j]].fnode[inc4(i)] == p[2*k]){
        v[2*k+1] = face[face[face[f].fface[k]].fface[j]].fnode[i=dec4(i)];
        v[2*k] = face[face[face[f].fface[k]].fface[j]].fnode[dec4(i)];
      }
      else{
        v[2*k+1] = face[face[face[f].fface[k]].fface[j]].fnode[i=inc4(i)];
      v[2*k] = face[face[face[f].fface[k]].fface[j]].fnode[inc4(i)];
    }
  }
  else{
    p[2*k] = face[face[f].fface[k]].fnode[j=inc4(j)];
    nk[k] = j;
    p[2*k+1] = face[face[f].fface[k]].fnode[inc4(j)];
    for(i=0 ; i <  4 ; i++){
      if(face[face[face[f].fface[k]].fface[j]].fnode[i] == p[2*k]) break;
    }
    if(face[face[face[f].fface[k]].fface[j]].fnode[inc4(i)] == p[2*k+1]){
      v[2*k] = face[face[face[f].fface[k]].fface[j]].fnode[i=dec4(i)];
      v[2*k+1] = face[face[face[f].fface[k]].fface[j]].fnode[dec4(i)];
    }
    else{
      v[2*k] = face[face[face[f].fface[k]].fface[j]].fnode[i=inc4(i)];
      v[2*k+1] = face[face[face[f].fface[k]].fface[j]].fnode[inc4(i)];
    }
  }
}

c = 0;
for(k=0 ; k < 4 ; k++){
  if((p[2*k+1] == p[2*(j=inc4(k))]) && (v[2*k+1] == p[2*j+1])){c = 1; break;}
}
if(c == 1){
  element[ci].enode[k+4] = p[2*k];
  element[ci].enode[j+4] = p[2*k+1];
  element[ci].enode[inc4(j)+4] = v[2*k+1];
  element[ci].enode[dec4(k)+4] = v[2*k];
  face[face[f].fface[k]].no = -1;
  cf[k] = -1;
  face[face[face[f].fface[k]].fface[nk[k]]].no = -1;
  top = 1;
  face[face[f].fface[j]].no = -1;
  cf[j] = -1;
  if((p[2*(j=inc4(j))] == v[2*k+1]) && (p[2*j+1] == v[2*k])){
    face[face[f].fface[j]].no = -1;
    cf[j] = -1;
  }
  if((p[2*j] != v[2*k+1]) && (p[2*j+1] == v[2*k])){
    joint_two_nodes(p[2*j], v[2*k+1]);
    p[2*j] = v[2*k+1];
    no = v[2*k+1];
    face[face[f].fface[j]].no = -1;
    cf[j] = -1;
  }
  if((p[2*j] == v[2*k+1]) && (p[2*j+1] != v[2*k])){
    joint_two_nodes(p[2*j+1], v[2*k]);
    p[2*j+1] = v[2*k];
    no = v[2*k];
    face[face[f].fface[j]].no = -1;
    cf[j] = -1;
  }
```

```
    if(cf[j] == 0){
      new_face(nf+1, face[f].fnode[j], face[f].fnode[inc4(j)],v[2*k],v[2*k+1]);
      cf[j] = 1;
    }
    if((p[2*(j=inc4(j))] == v[2*k]) && (p[2*j+1] == p[2*k])){
      face[face[f].fface[j]].no = -1;
      cf[j] = -1;
    }
    if((p[2*j] != v[2*k]) && (p[2*j+1] == p[2*k])){
      joint_two_nodes(p[2*j], v[2*k]);
      p[2*j] = v[2*k];
      no = v[2*k];
      face[face[f].fface[j]].no = -1;
      cf[j] = -1;
    }
    if((p[2*j] == v[2*k]) && (p[2*j+1] != p[2*k])){
      joint_two_nodes(p[2*j+1], p[2*k]);
      p[2*j+1] = p[2*k];
      no = p[2*k];
      face[face[f].fface[j]].no = -1;
      cf[j] = -1;
    }
    if(cf[j] == 0){
      new_face(nf+1, face[f].fnode[j], face[f].fnode[k], p[2*k], v[2*k]);
      cf[j] = 1;
    }
  }

  if((cf[0] == 0) && (cf[1] == 0) && (cf[2] == 0) && (cf[3]  == 0)){
    if((p[1] == p[2]) && ((face[f].a[0] <= a135) || (face[f].a[1] <= a135))){
      element[ci].enode[4] = p[0];
      element[ci].enode[5] = p[1];
      element[ci].enode[6] = p[3];
      cf[0] = -1;
      face[face[f].fface[0]].no = -1;
      cf[1] = -1;
      face[face[f].fface[1]].no = -1;
    }
    if((p[3] == p[4]) && ((face[f].a[1] <= a135) || (face[f].a[2] <= a135))){
      if(cf[1] != 0){
        element[ci].enode[7] = p[5];
        cf[2] = -1;
        face[face[f].fface[2]].no = -1;
      }
      else{
        element[ci].enode[5] = p[2];
        element[ci].enode[6] = p[3];
        element[ci].enode[7] = p[5];
        cf[1] = -1;
        face[face[f].fface[1]].no = -1;
        cf[2] = -1;
        face[face[f].fface[2]].no = -1;
      }
    }
```

```
      if((p[5] == p[6]) && ((face[f].a[2] <= a135) || (face[f].a[3] <= a135))){
        if(cf[2] != 0){
          element[ci].enode[4] = p[7];
          cf[3] = -1;
          face[face[f].fface[3]].no = -1;
        }
        else{
          element[ci].enode[6] = p[4];
          element[ci].enode[7] = p[5];
          element[ci].enode[4] = p[7];
          cf[2] = -1;
          face[face[f].fface[2]].no = -1;
          cf[3] = -1;
          face[face[f].fface[3]].no = -1;
        }
      }
      if((p[7] == p[0]) && ((face[f].a[3] <= a135) || (face[f].a[0] <= a135))){
        if((cf[3] != 0) && (cf[1] == 0)){
          element[ci].enode[5] = p[1];
          cf[0] = -1;
          face[face[f].fface[0]].no = -1;
        }
        if((cf[3] == 0) && (cf[1] == 0)){
          element[ci].enode[7] = p[6];
          element[ci].enode[4] = p[7];
          element[ci].enode[5] = p[1];
          cf[3] = -1;
          face[face[f].fface[3]].no = -1;
          cf[0] = -1;
          face[face[f].fface[0]].no = -1;
        }
        if((cf[3] == 0) && (cf[1] != 0)){
          element[ci].enode[7] = p[6];
          cf[3] = -1;
          face[face[f].fface[3]].no = -1;
        }
      }
    }

    if((cf[0] == 0) && (cf[1] == 0) && (cf[2] == 0) && (cf[3]  == 0)){
      c = -1;
      aux = 10.0;
      for(k=0 ; k < 4 ; k++){
        if((face[f].a[k] <= a135) && (face[face[f].fface[k]].a[nk[k]] <= a135) &&
           ((face[f].a[k] + face[face[f].fface[k]].a[nk[k]]) < aux)){
          aux = face[f].a[k] + face[face[f].fface[k]].a[nk[k]];
          c = k;
        }
      }
      if(c != -1){
        face[face[f].fface[c]].no = -1;
        face[face[face[f].fface[c]].fface[nk[c]]].no = -1;
        top = 1;
        cf[c] = -1;
      element[ci].enode[c+4] = p[2*c];
      element[ci].enode[(j=inc4(c))+4] = p[2*c+1];
      element[ci].enode[(j=inc4(j))+4] = v[2*c+1];
      new_face(nf+1,face[f].fnode[inc4(c)],face[f].fnode[j],v[2*c+1],p[2*c+1]);
      cf[inc4(c)] = 1;
      element[ci].enode[inc4(j)+4] = v[2*c];
      if((p[2*j] == v[2*c+1]) && (p[2*j+1] == v[2*c])) {
        face[face[f].fface[j]].no= -1;
        cf[j] = -1;
      }
      else{
        cf[j] = 1;
        new_face(nf+1, face[f].fnode[j], face[f].fnode[j=inc4(j)], v[2*c],
                 v[2*c+1]);
      }
      cf[j] = 1;
      new_face(nf+1,face[f].fnode[j],face[f].fnode[c],p[2*c],v[2*c]);
    }
  }

  for(k=0 ; k < 4 ; k++){
    if((cf[k]==0)&&(cf[dec4(k)]==0)&&(cf[j=inc4(k)]==0)&&(face[f].a[k]<=a135)){
      cf[k] = -1;
      face[face[f].fface[k]].no = -1;
      element[ci].enode[k+4] = p[2*k];
      element[ci].enode[j+4] = p[2*k+1];
    }
  }

  if(cf[0] == 0){
    if((cf[3] != 0) && (cf[1] != 0)){
      cf[0] = 1;
      new_face(nf+1, face[f].fnode[0], face[f].fnode[1], element[ci].enode[5],
               element[ci].enode[4]);
    }
    if((cf[3] == 0) && (cf[1] != 0)){
      element[ci].enode[4] = projection(f, face[f].fnode[0]);
      cf[0] = 1;
      new_face(nf+1, face[f].fnode[0], face[f].fnode[1], element[ci].enode[5],
               element[ci].enode[4]);
    }
    if((cf[3] != 0) && (cf[1] == 0)){
      element[ci].enode[5] = projection(f, face[f].fnode[1]);
      cf[0] = 1;
      new_face(nf+1, face[f].fnode[0], face[f].fnode[1], element[ci].enode[5],
               element[ci].enode[4]);
    }
    if((cf[3] == 0) && (cf[1] == 0)){
      element[ci].enode[4] = projection(f, face[f].fnode[0]);
      element[ci].enode[5] = projection(f, face[f].fnode[1]);
      cf[0] = 1;
      new_face(nf+1, face[f].fnode[0], face[f].fnode[1], element[ci].enode[5],
               element[ci].enode[4]);
    }
```

```
}
if(cf[1] == 0){
  if(cf[2] != 0){
    cf[1] = 1;
    new_face(nf+1, face[f].fnode[1], face[f].fnode[2], element[ci].enode[6],
            element[ci].enode[5]);
  }
  else{
    element[ci].enode[6] = projection(f, face[f].fnode[2]);
    cf[1]= 1;
    new_face(nf+1, face[f].fnode[1], face[f].fnode[2], element[ci].enode[6],
            element[ci].enode[5]);
  }
}
if(cf[2] == 0){
  if(cf[3] != 0){
    cf[2] = 1;
    new_face(nf+1, face[f].fnode[2], face[f].fnode[3], element[ci].enode[7],
            element[ci].enode[6]);
  }
  else{
    element[ci].enode[7] = projection(f, face[f].fnode[3]);
    cf[2] = 1;
    new_face(nf+1, face[f].fnode[2], face[f].fnode[3], element[ci].enode[7],
            element[ci].enode[6]);
  }
}
if(cf[3] == 0){
  cf[3] = 1;
  new_face(nf+1, face[f].fnode[3], face[f].fnode[0], element[ci].enode[4],
          element[ci].enode[7]);
}

if(top == 0) new_face(nf+1, element[ci].enode[4], element[ci].enode[5],
                    element[ci].enode[6], element[ci].enode[7]);
face[f].no = -1;
c = -1;
i = 0;
for(k=0 ; k < 4 ; k++){
  if((cf[k] == 1) && (cf[dec4(k)] == 1) && (cf[j=inc4(k)] == 1) &&
     (cf[j=inc4(j)] == -1)) {c = k; i = 1; break;}
  if((cf[k] == 1) && (cf[dec4(k)] == -1) && (cf[j=inc4(k)] == 1) &&
     (cf[j=inc4(j)] == -1)) {c = k; i = 2; break;}
}
if(i == 1) smooth_one_face(c, j);
if(i == 2) smooth_two_faces(c, j);
ads_command(RTSTR, "3dface", RT3DPOINT, node[element[ci].enode[0]].coord,
           RT3DPOINT, node[element[ci].enode[1]].coord,
           RT3DPOINT, node[element[ci].enode[2]].coord,
           RT3DPOINT, node[element[ci].enode[3]].coord,
           RTSTR, "", 0);
ads_command(RTSTR, "3dface", RT3DPOINT, node[element[ci].enode[0]].coord,
           RT3DPOINT, node[element[ci].enode[1]].coord,
           RT3DPOINT, node[element[ci].enode[5]].coord,
          RT3DPOINT, node[element[ci].enode[4]].coord,
          RTSTR, "", 0);
ads_command(RTSTR, "3dface", RT3DPOINT, node[element[ci].enode[1]].coord,
          RT3DPOINT, node[element[ci].enode[2]].coord,
          RT3DPOINT, node[element[ci].enode[6]].coord,
          RT3DPOINT, node[element[ci].enode[5]].coord,
          RTSTR, "", 0);
ads_command(RTSTR, "3dface", RT3DPOINT, node[element[ci].enode[2]].coord,
          RT3DPOINT, node[element[ci].enode[3]].coord,
          RT3DPOINT, node[element[ci].enode[7]].coord,
          RT3DPOINT, node[element[ci].enode[6]].coord,
          RTSTR, "", 0);
ads_command(RTSTR, "3dface", RT3DPOINT, node[element[ci].enode[3]].coord,
          RT3DPOINT, node[element[ci].enode[0]].coord,
          RT3DPOINT, node[element[ci].enode[4]].coord,
          RT3DPOINT, node[element[ci].enode[7]].coord,
          RTSTR, "", 0);
ads_command(RTSTR, "3dface", RT3DPOINT, node[element[ci].enode[4]].coord,
          RT3DPOINT, node[element[ci].enode[5]].coord,
          RT3DPOINT, node[element[ci].enode[6]].coord,
          RT3DPOINT, node[element[ci].enode[7]].coord,
          RTSTR, "", 0);
for(c=0 ; c < lafi; c++){
  set_adjacent_faces(fi-c-1);
  for(i=0 ; i < 4 ; i++) set_adjacent_faces(face[fi-c-1].fface[i]);
  for(i=0 ; i < 4 ; i++){
    face[fi-c-1].a[i] = face_angles(face[fi-c-1].no, i);
    for(j=0 ; j < 4 ; j++)
      face[face[fi-c-1].fface[i]].a[j] =
                      face_angles(face[face[fi-c-1].fface[i]].no, j);
  }
}
if(no != -1){
  for(i=0 ; i < node[no].nfi ; i++){
    if(face[node[no].nface[i]].no != -1){
      set_adjacent_faces(node[no].nface[i]);
      for(k=0 ; k < 4 ; k++){
        face[node[no].nface[i]].a[k] = face_angles(node[no].nface[i], k);
      }
    }
  }
}
if((cf[0] == -1) && (cf[1] == -1) && (cf[2] == -1) && (cf[3] == -1)){
  for(k=0 ; k < 8 ; k++){
    for(i=0 ; i < node[element[ci].enode[k]].nfi ; i++){
      if(face[node[element[ci].enode[k]].nface[i]].no != -1){
        set_adjacent_faces(node[element[ci].enode[k]].nface[i]);
        for(j=0 ; j < 4 ; j++){
          face[node[element[ci].enode[k]].nface[i]].a[j] =
                    face_angles(node[element[ci].enode[k]].nface[i], j);
        }
      }
    }
  }
  }
  ci++;
}
```

A.23 Function *advance_face_front(int nf)*

```
int advance_face_front(int nf)
{
  int i, k, s=0, w=-1;
  aci = 0;
  afi = 0;
  tfi = fi;
  for(i=0 ; i < tfi ; i++){
    if((face[i].ord == nf) && (face[i].no != -1)){
      w = 0;
      for(k=0 ; k < 4 ; k++){
        if(face[i].a[k] <= a135) {s = i; w = 1; break;}
      }
    }
    if(w == 1) break;
  }
  if(w == -1) return 0;
  create_element(nf, s);
  aci++;
  i = inct(s);
  while(i != s){
    if(face[i].no != -1) {
      create_element(nf, i);
      aci++;
    }
    i = inct(i);
  }
  return afi;
}
```

A.24 Function *smooth_3D()*

```
void smooth_3D()
{
  int i, ii, j, k, kk, an, as;
  POINT P;
  printf("   smooth_3D\n");
  as = ci / 5;
  for(ii=0 ; ii < as ; ii++){
    for(j=pn ; j < ni ; j++){
      if(node[j].no != -1){
        an = 0;
        P[X] = 0.0;
        P[Y] = 0.0;
        P[Z] = 0.0;
        for(i=0 ; i < ci ; i++){
          for(k=0 ; k < 8 ; k++){
            if(element[i].enode[k] == j){
              for(kk=0 ; kk < 8 ; kk++){
                if(kk != k){
                  P[X] = P[X] + node[element[i].enode[kk]].coord[X];
                  P[Y] = P[Y] + node[element[i].enode[kk]].coord[Y];
                  P[Z] = P[Z] + node[element[i].enode[kk]].coord[Z];
                  an++;
                }
              }
            }
          }
        }
        node[j].coord[X] = P[X] / an;
        node[j].coord[Y] = P[Y] / an;
        node[j].coord[Z] = P[Z] / an;
      }
    }
  }
  reprint_element(0);
}
```

A.25 Auxiliary functions to check the 3D-mesh

```
/* VOLUME OF TETRAHEDRON */
REAL volume_of_tetrahedron(REAL *P0, REAL *P1, REAL *P2, REAL *P3)
{
  POINT v, v0, v1, v3;
  v0[X] = P1[X] - P0[X];
  v0[Y] = P1[Y] - P0[Y];
  v0[Z] = P1[Z] - P0[Z];
  v1[X] = P2[X] - P0[X];
  v1[Y] = P2[Y] - P0[Y];
  v1[Z] = P2[Z] - P0[Z];
  v3[X] = P3[X] - P0[X];
  v3[Y] = P3[Y] - P0[Y];
  v3[Z] = P3[Z] - P0[Z];
  vector_product(v0, v1, v);
  return fabs(inner_product(v, v3) / 6.0);
}

/* SET NODE TYPE IN 2D-MESH */
void set_node_type_face()
{
  int i=0, j, ii, w, no[15], k, k0, k1, s=1, i0, i1;
  for( ; ; ){
    if((i == 0) && (s == 0)) break;
    if(i == 0) s = 0;
    if((node[i].type == -1) && (node[i].no != -1)){
      w = 0;
      for(j=0 ; j < node[i].nfi ; j++){
        for(k=0 ; k < 4 ; k++) if(face[node[i].nface[j]].fnode[k] == i) break;
        k0 = dec4(k);
        k1 = inc4(k);
        i0 = 0;
        i1 = 0;
        for(ii=0 ; ii < w ; ii++){
          if(no[ii] == face[node[i].nface[j]].fnode[k0]) i0 = 1;
          if(no[ii] == face[node[i].nface[j]].fnode[k1]) i1 = 1;
        }
        if(i0 == 0){
          no[w] = face[node[i].nface[j]].fnode[k0];
          w++; if(w > 15)printf("     ERROR-3\n");
        }
        if(i1 == 0){
          no[w] = face[node[i].nface[j]].fnode[k1];
          w++; if(w > 15)printf("     ERROR-4\n");
        }
      }
      for(j=0 ; j < w ; j++){
        if(node[no[j]].type != -1){
          if(node[no[j]].type == 0) node[i].type = 1;
          if(node[no[j]].type == 1) node[i].type = 0;
          s = 1;
          break;
        }
      }
    }
    i = incn(i);
  }
}

/* SET NODE TYPE IN 3D-MESH */
void set_node_type_element()
{
  int i=0, j, ii, w, no[15], q, e[15], k, k0, k1, k2, s=1, i0, i1, i2;
  for( ; ; ){
    if((i == 0) && (s == 0)) break;
    if(i == 0) s = 0;
    q = 0;
    if((node[i].type == -1) && (node[i].no != -1)){
      for(j=0 ; j < ci ; j++){
        for(k=0 ; k < 8 ; k++){
          if(element[j].enode[k] == i){
            e[q] = j;
            q++; if(q > 15)printf("     ERROR-88\n");
            break;
          }
        }
      }
      w = 0;
      for(j=0 ; j < q ; j++){
        for(k=0 ; k < 8 ; k++) if(element[e[j]].enode[k] == i) break;
        k0 = k - 1;
        if(k == 0) k0 = 3;
        if(k == 4) k0 = 7;
        k1 = k + 1;
        if(k == 3) k0 = 0;
        if(k == 7) k0 = 4;
        k2 = k + 4;
        if(k > 3) k2 = k - 4;
        i0 = 0;
        i1 = 0;
        i2 = 0;
        for(ii=0 ; ii < w ; ii++){
          if(no[ii] == element[e[j]].enode[k0]) i0 = 1;
          if(no[ii] == element[e[j]].enode[k1]) i1 = 1;
          if(no[ii] == element[e[j]].enode[k2]) i2 = 1;
        }
        if(i0 == 0){
          no[w] = element[e[j]].enode[k0];
          w++; if(w > 15)printf("     ERROR-5\n");
        }
        if(i1 == 0){
          no[w] = element[e[j]].enode[k1];
          w++; if(w > 15)printf("     ERROR-6\n");
        }
        if(i2 == 0){
          no[w] = element[e[j]].enode[k2];
          w++; if(w > 15)printf("     ERROR-7\n");
        }
      }
      for(j=0 ; j < w ; j++){
        if(node[no[j]].type != -1){
          if(node[no[j]].type == 0) node[i].type = 1;
          if(node[no[j]].type == 1) node[i].type = 0;
          s = 1;
          break;
        }
      }
    }
    i = incn(i);
  }
}

/* CHECK OF ELEMENT TYPE */
int element_type(int e)
{
  int t0=0, t1=0;
  if((node[element[e].enode[0]].type == 1) &&
     (node[element[e].enode[2]].type == 1) &&
     (node[element[e].enode[5]].type == 1) &&
     (node[element[e].enode[7]].type == 1)) t0 = 1;
  if((node[element[e].enode[1]].type == 1) &&
     (node[element[e].enode[3]].type == 1) &&
     (node[element[e].enode[4]].type == 1) &&
     (node[element[e].enode[6]].type == 1)) t1 = 1;
  if(t0 == t1) return -1;
  if((t0 == 1) && (t1 == 0)) return 0;
  if((t0 == 0) && (t1 == 1))  return 1;
}

/* VOLUME OF ELEMENT */
REAL volume_of_element(int e)
{
  int type;
  REAL vol;
  type = element_type(e);
  if(type == -1) return (-1E+10);
  if(type == 0){
    vol = volume_of_tetrahedron(node[element[e].enode[0]].coord,
                                node[element[e].enode[1]].coord,
                                node[element[e].enode[2]].coord,
                                node[element[e].enode[5]].coord) +
          volume_of_tetrahedron(node[element[e].enode[0]].coord,
                                node[element[e].enode[2]].coord,
                                node[element[e].enode[3]].coord,
                                node[element[e].enode[7]].coord) +
          volume_of_tetrahedron(node[element[e].enode[0]].coord,
                                node[element[e].enode[4]].coord,
                                node[element[e].enode[5]].coord,
                                node[element[e].enode[7]].coord) +
          volume_of_tetrahedron(node[element[e].enode[2]].coord,
                                node[element[e].enode[5]].coord,
                                node[element[e].enode[6]].coord,
                                node[element[e].enode[7]].coord) +
          volume_of_tetrahedron(node[element[e].enode[0]].coord,
                                node[element[e].enode[2]].coord,
                                node[element[e].enode[5]].coord,
                                node[element[e].enode[7]].coord);
  }
  if(type == 1){
    vol = volume_of_tetrahedron(node[element[e].enode[0]].coord,
                                node[element[e].enode[1]].coord,
                                node[element[e].enode[3]].coord,
                                node[element[e].enode[4]].coord) +
          volume_of_tetrahedron(node[element[e].enode[1]].coord,
                                node[element[e].enode[2]].coord,
                                node[element[e].enode[3]].coord,
                                node[element[e].enode[6]].coord) +
          volume_of_tetrahedron(node[element[e].enode[1]].coord,
                                node[element[e].enode[4]].coord,
                                node[element[e].enode[5]].coord,
                                node[element[e].enode[6]].coord) +
          volume_of_tetrahedron(node[element[e].enode[3]].coord,
                                node[element[e].enode[4]].coord,
                                node[element[e].enode[6]].coord,
                              node[element[e].enode[7]].coord) +
          volume_of_tetrahedron(node[element[e].enode[1]].coord,
                              node[element[e].enode[3]].coord,
                              node[element[e].enode[4]].coord,
                              node[element[e].enode[6]].coord);
  }
  return vol;
}

/* VOLUME OF FACE */
REAL volume_of_face(int f)
{
  int k, k0, k1, k2;
  REAL vol=0, d;
  POINT P[8], v0, v1, v;
  P[4][X] = node[face[f].fnode[0]].coord[X];
  P[4][Y] = node[face[f].fnode[0]].coord[Y];
  P[4][Z] = 0.0;
  P[5][X] = node[face[f].fnode[1]].coord[X];
  P[5][Y] = node[face[f].fnode[1]].coord[Y];
  P[5][Z] = 0.0;
  P[6][X] = node[face[f].fnode[2]].coord[X];
  P[6][Y] = node[face[f].fnode[2]].coord[Y];
  P[6][Z] = 0.0;
  P[7][X] = node[face[f].fnode[3]].coord[X];
  P[7][Y] = node[face[f].fnode[3]].coord[Y];
  P[7][Z] = 0.0;
  if(node[face[f].fnode[0]].type == 1) k = 0;
  else k = 1;
  k0 = dec4(k);
  k1 = inc4(k);
  k2 = inc4(k1);
  vector_no(face[f].fnode[k], face[f].fnode[k1], v0);
  vector_no(face[f].fnode[k], face[f].fnode[k2], v1);
  vector_product(v0, v1, v);
  d = vector_norm(v);
  scalar_vector(1.0/d, v);
  d = inner_product(face[f].norm, v);
  if(d < 0) scalar_vector(-1.0, v);
  if(v[Z] != 0){
    vol = volume_of_tetrahedron(node[face[f].fnode[k]].coord, P[k+4],
                              P[k1+4], P[k2+4]) +
          volume_of_tetrahedron(node[face[f].fnode[k1]].coord,
                              node[face[f].fnode[k]].coord,
                              node[face[f].fnode[k2]].coord, P[k2+4]) +
          volume_of_tetrahedron(node[face[f].fnode[k]].coord,
                              node[face[f].fnode[k1]].coord, P[k1+4],
                              P[k2+4]);
    if(v[Z] > 0) vol = -vol;
  }
  k1 = dec4(k);
  vector_no(face[f].fnode[k], face[f].fnode[k1], v0);
  vector_no(face[f].fnode[k], face[f].fnode[k2], v1);
  vector_product(v0, v1, v);
  d = vector_norm(v);
  scalar_vector(1.0/d, v);
  d = inner_product(face[f].norm, v);
  if(d < 0) scalar_vector(-1.0, v);
  if(v[Z] != 0){
    d = volume_of_tetrahedron(node[face[f].fnode[k]].coord, P[k+4], P[k1+4],
                              P[k2+4]) +
        volume_of_tetrahedron(node[face[f].fnode[k1]].coord,
                              node[face[f].fnode[k]].coord,
                              node[face[f].fnode[k2]].coord, P[k2+4]) +
        volume_of_tetrahedron(node[face[f].fnode[k]].coord,
                              node[face[f].fnode[k1]].coord, P[k1+4], P[k2+4]);
    if(v[Z] > 0) d = -d;
    vol = vol + d;
  }
  return vol;
}

/* VOLUME OF 2D-MESH */
REAL volume_of_2D_mesh(int j)
{
  int i;
  REAL vol=0;
  for(i=0 ; i < fi ; i++){
    if((face[i].ord == j) && (face[i].no != -1)) vol = vol + volume_of_face(i);
  }
  return vol;
}

/* VOLUME OF 3D-MESH */
REAL volume_of_3D_mesh(int s, int e)
{
  int i;
  REAL vol=0;
  for(i=s ; i < e ; i++) vol = vol + volume_of_element(i);
  return vol;
}

/* CHECK IF THERE ARE TWO AND ONLY TWO ELEMENTS PER FACE */
void check_two_elements_per_face()
{
  int i, j, k, ii, q, w;
  for(i=0 ; i < fi ; i++){
    if(face[i].ord != 0){
      q = 0;
      for(j=0 ; j < ci ; j++){
        w = 0;
        for(k=0 ; k < 8 ; k++){
          for(ii=0 ; ii < 4 ; ii++){
            if(element[j].enode[k] == face[i].fnode[ii]) w++;
          }
        }
        if(w == 4) q++;
      }
      if(q != 2) printf("  3D-MESH ERROR in face=%d\n", i);
    }
  }
}
```

A.26 *ADS* template

```
static int loadfuncs();
int geo();
int autoqm();
int mesh3D();
void main(int argc, char *argv[])
{
  int stat;
  short scode = RSRSLT;
  ads_init(argc, argv);
  ap_init();
  for ( ;; ) {
    if ((stat = ads_link(scode)) < 0) {
      printf("TEMPLATE: bad status from ads_link() = %d\n", stat);
      exit(1);
    }
    scode = RSRSLT;
    switch (stat) {
    case RQXLOAD:
      scode = loadfuncs() ?  RSRSLT : RSERR;
      break;
    default:  break;
    }
  }
}
static int loadfuncs()
{
  int a=0, b=0, c=0;
  if(ads_defun("C:geo", 0) == RTNORM){
    ads_regfunc(geo, 0);
    b = 1;
  }
  if(ads_defun("C:autoqm", 1) == RTNORM){
    ads_regfunc(autoqm, 1);
    a = 1;
  }
  if(ads_defun("C:mesh3D", 2) == RTNORM){
    ads_regfunc(mesh3D, 2);
    c = 1;
  }
  return (a * b * c);
}
```